FAKE PHONG SHADING

by

Daniel Vlasic

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 17, 2002
CJune2002

Author _____
Department of Electrical Engineering and Computer Science
May 17, 2002

Approved by _____
Leonard McMillan
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

FAKE PHONG SHADING

by Daniel Vlasic

Submitted to the
Department of Electrical Engineering and Computer Science

May 17, 2002

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
And Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

In the real-time 3D graphics pipeline framework, rendering quality greatly depends on illumination and shading models. The highest-quality shading method in this framework is Phong shading. However, due to the computational complexity of Phong shading, current graphics hardware implementations use a simpler Gouraud shading. Today, programmable hardware shaders are becoming available, and, although real-time Phong shading is still not possible, there is no reason not to improve on Gouraud shading.

This thesis analyzes four different methods for approximating Phong shading: quadratic shading, environment map, Blinn map, and quadratic Blinn map. Quadratic shading uses quadratic interpolation of color. Environment and Blinn maps use texture mapping. Finally, quadratic Blinn map combines both approaches, and quadratically interpolates texture coordinates.

All four methods adequately render higher-resolution methods. However, only Blinn map and quadratic Blinn map provide reasonable quality on coarser meshes. Moreover, quadratic Blinn map is not implementable in current hardware shaders. Therefore, Blinn map is the best presented Phong approximation method. It renders in real-time, with near-Phong quality, and easily integrates into the 3D graphics pipeline.

Thesis Supervisor: Leonard McMillan
Title: Associate Professor, MIT Department of EECS

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Leonard McMillan, for support and opportunity to work on 'Fake Phong Shading' project. He provided me with guidance and patience, as well as all the flexibility and freedom while tackling this problem. I would also like to thank my labmates for keeping a great atmosphere in the graphics lab, filled with hours of hardcore research and console game-playing. Finally, I am most grateful to my family and Ana for always backing me up, helping me get where I am now in life.

And, of course, I should not have forgotten the one person that holds this lab together, the one who provides essential nutrition to all of us (but mostly me) – our secretary, Bryt Bradley!

Chapter 1

INTRODUCTION

Real-time 3D graphics is the most researched and developed branch of computer graphics. It is so commonplace that dedicated hardware has been developed to support it on almost all of today's available computers. It wasn't always like this: the early work on 3D rendering and hardware was done on high-end computers and workstations.

Recently, 3D graphics has migrated to personal computers with dedicated hardware. With the advent of video games industry, the hardware has been pushed to render increasingly complex scenes at interactive rates (>30 frames per second). In order to achieve this, modern graphics cards resort to lower-quality rendering techniques that are approximations to the well established, and much slower, high-quality methods. Hardware vendors have widely adopted a standardized graphics pipeline for real-time rendering, but thanks to the flexibility of the hardware, the most recent architectures enable programmers to implement algorithms that improve rendering quality at very little or no reduction in rendering speed.

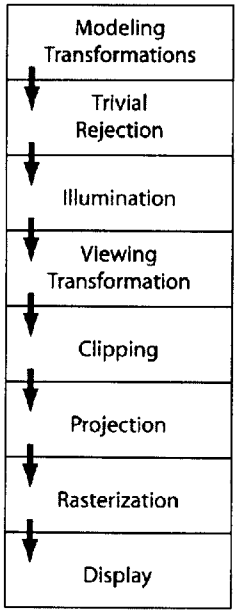

Figure 1.1 Standard graphics pipeline

In the classic interactive graphics pipeline (Figure 1.1) 3D objects are represented by triangular meshes. In addition to triangles approximating the object's geometry, a triangular mesh contains material information as well as the surface normals at triangle vertices. The rendering pipeline describes a number of processing steps needed to generate the resulting image on the screen. First, modeling transformations appropriately orient models, defined in their own *model space*, in a common coordinate frame referred to as the *world space*. Next, those objects that cannot possibly be seen are eliminated by trivial rejection. The remaining objects are thereafter illuminated and each vertex is assigned a color. After illumination, viewing transformation maps the points from world space into camera's point of view – *eye space*. There, objects are clipped against the boundaries of the final visible volume called *viewing frustum*. At that point, the computation transitions from three to two dimensions by projecting the viewing frustum onto a plane called *screen space*. Finally, with rasterization, the objects are scan-converted to pixels in a screen buffer, which in turn can be displayed.

The quality of the appearance of objects rendered using the pipeline greatly depends on illumination and shading stages. My research focuses on those two highly coupled phases. As my thesis I will implement and evaluate several popular real-time shading methods for approximating high-quality 3D rendering, all of which can make use of the power and flexibility of current graphics hardware. In addition, I will introduce and evaluate a few new techniques. I will analyze all of these methods assessing their quality and efficiency. Finally, I will compare them both analytically and qualitatively to the standard high-quality rendering methods.

The central claims of my thesis are that modern programmable graphics hardware can be used to approximate significantly higher quality renderings than are typically available through the classic pipeline approach. Moreover, this hardware can be used to closely approximate the highest quality per-pixel illumination algorithms, which previously have been used only in off-line (non-realtime) rendering algorithms. I will explore two techniques for achieving higher quality interactive renderings using existing programmable hardware. First, I will explore the implementation of higher-order interpolants in the "pixel shader" phase of rendering. Next, I will evaluate the use of modern texture mapping hardware to improve the illumination approximations used.

The remainder of the thesis is organized as follows: In chapter 2, I discuss previous work in the area of real-time rendering. In chapter 3, I explain the background information on the topic. Chapter 4 describes different interpolation methods for high-quality rendering approximation. Chapter 5 deals with rendering by means of texture mapping. In chapter 6, I introduce quadratic interpolation of texture coordinates for improving the rendering quality. Chapter 7 presents the results and provides analysis of all the mentioned methods. In

chapter 8, I discuss the results and possible future work, as well as conclude the thesis.

Chapter 2

PREVIOUS WORK

In this chapter I discuss the work on real-time 3D rendering done during the last few decades. This includes: illumination models, interpolation techniques, and texture mapping.

Much of the early work on real-time lighting and shading was done in University of Utah in the early seventies. Henry Gouraud created one of the first algorithms for smoothly shading curved surfaces represented by triangular meshes [1]. He proposed computing the approximate lighting at only a few points per triangle, the triangle vertices, and then interpolating those colors across the surface of the triangle. Although he considered various types of interpolation (quadratic, cubic), the common Gouraud shading algorithm is synonymous to linear interpolation of color. Accordingly, it can be expressed incrementally and implemented very efficiently in hardware; every modern 3D graphics chip supports it.

The lighting model used by Gouraud expressed only the diffuse or Lambertian reflectance, which depends on the orientation of the surface relative to the incoming light. This simple lighting model cannot represent shiny and reflective materials. Subsequently, Phong introduced a more elaborate phenomenological-based lighting model [2]. He added a model for specularity – the reflection of the light source on the material – to the lighting equation. According to this improved model, in addition to the surface normal and incoming light, the specular light intensity is also dependent on the viewer position. Phong developed an analytic expression for heuristically modeling a range of surfaces between the diffuse and mirror-like by adding a term that controls the spread of

the specular highlight, which is often referred to as the material's shininess. The lighting equation in this form is known as Phong illumination. Phong also improved Gouraud shading by pointing out that rather than interpolating the color across a triangle, one should interpolate the surface normal and re-evaluate the lighting equation at each pixel – an algorithm known as Phong shading.

Later in the seventies, Jim Blinn published a variant to Phong's lighting equation [3]. His version was based on Torrence-Sparrow explorations of modeling real light, which made it physically more correct than Phong's. Their theory states that on the micro-level every material is composed of numerous mirroring facets with some probabilistic distribution of orientation. Blinn's insight was to measure how much the average surface normal deviates from the "ideal normal" that would reflect all light towards the viewer. According to Snell's law of reflection, this ideal normal is the bisector vector between light and viewer direction, also called the half-vector. In view of that, Blinn modeled specular highlights to be strongest when the normal and half-vector coincide, and to fall off as they diverge. Up to this day, graphics hardware most frequently models lighting by Blinn's variant of Phong's illumination equation.

In a standard graphics pipeline, the highest quality rendering is achieved by rendering polygonal meshes using Blinn's illumination model in tandem with Phong shading. However, even the latest commodity hardware does not apply Phong shading at interactive frame rates: they still use Gouraud shading. As a result, programmers (mostly game developers) have come up with, and are still developing, various methods of approximating Phong shading. Those methods fall in between Gouraud and Phong shading both in speed and in rendering quality, and the programmers' challenge is to find the best trade-off between quality and performance (at least until the hardware becomes strong enough for Phong).

Some methods of improving linear shading were mentioned even by Gouraud – higher order interpolation. Quadratic and cubic shading were considered and implemented by several researchers [5]. Both of these higher-order methods can express extremes and can be continuous across triangles, which makes them suitable for shading curved surfaces. On the other hand, higher order shading is more computationally intensive than linear Gouraud, both in the initialization and per-pixel computation. Furthermore, since a quadratic can have only one local extreme it cannot show more than one highlight per triangle.

A Phong approximation technique called "Fast Phong Shading" was introduced by Bishop and Weimer [4] in mid-eighties. They approximate true Phong shading by expanding the illumination equation using two-dimensional form of Taylor's series – expanding to the second degree about the triangle centroid. Conveniently, the resulting expression (a quadratic) can be evaluated using forward differencing, which makes it suitable for hardware implementation. Although fast Phong shading improves the speed over standard Phong shading, it does not enforce continuity across triangle edges. Furthermore, the error of the approximation grows with the triangle size since the Taylor series is expanded about the centroid. In addition, to simplify the computation, Bishop and Weimer assumed that both the eye and the light sources are infinitely far away from the object. This makes fast Phong shading inappropriate for today's uses of real-time 3D graphics such as computer games and realistic walkthroughs and visualizations.

For the last decade, real-time desktop 3D graphics on has been driven by the progress of computer gaming industry. One of the most important advances brought by computer games is real-time texture mapping. Texture mapping is a very powerful tool developed for expressing material properties that lighting models cannot simulate, for example high-frequency detail and patterns. The

efficiency and flexibility of texture-mapping hardware led programmers to investigate using textures for lighting and shading as opposed to analytic models. As it turns out, with appropriate textures and texture-mapping algorithms it is possible to produce a reasonable approximation to Phong shading. Today there are a number of such algorithms employed primarily in games. One of these is "environment mapping" introduced by Blinn and Newel in [6] and intended for simulating global reflection. Environment mapping essentially wraps an image of the environment as seen from object's point of view around the object. The environment is mapped onto object's surface by using vertex normals and/or view vector to index into the texture. Presently there are several standard techniques used for environment mapping, such as spherical, cubic, and bi-quadratic mapping.

There are several other methods that fall in between Gouraud and Phong shading that I will not discuss since they have not achieved a wide acceptance. They include techniques such as interpolating angles and cosines of angles. Some higher-level methods, such as bi-cubic patches and Bezier triangles, are not implemented in hardware and fall out of the scope of this research.

The last 35 or so years have seen significant innovation in real-time 3D rendering, from simple flat shading to complex texture mapping. Many of the classic approaches, such as Gouraud shading, are well established and supported in modern hardware; they have seen little change since they were first invented. Textures have introduced lots of variations and improvements to the classic algorithms, due to the flexibility of their use. What's more, with the emergence of programmable components of the pipeline, we are given tools to implement new and completely different rendering algorithms. In this research I plan to evaluate and extend many of these techniques that can use the power of hardware to better approximate Phong shading; methods I classify as "fake Phong

shading". In the next chapter I will introduce the background information essential to understanding the remainder of the thesis where I describe and analyze these methods.

Chapter 3

BACKGROUND

In this chapter I discuss the work on real-time 3D rendering done during the last few decades. Much of the graphics pipeline can be expressed using simple linear algebra, though it does contain some more complex techniques. In this chapter I define the conventions used in the remainder of the thesis. Furthermore, I explain some of the necessary graphic-specific techniques, namely: illumination equations, texture mapping, triangle parameterizations, and forward differencing.

3.1 Conventions

Most of the calculations in this thesis rely on a set of normalized vectors defined at the surface of each triangle. They describe the lighting environment of triangles and are essential for illumination calculation. They are depicted in the following figure:

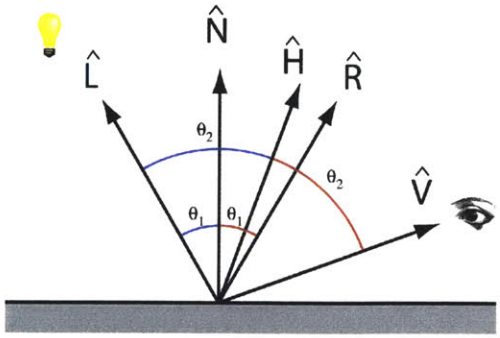

Figure 3.1 Conventional vectors: $\hat{N}$, the surface normal; $\hat{L}$, the unit vector from the surface point to a light source; $\hat{V}$, the unit vector from the surface point to the viewer's eye; $\hat{R}$, the unit vector in the direction of the reflected light; $\hat{H}$, the half-vector between $\hat{L}$ and $\hat{V}$

$\hat{N}$, $\hat{L}$, and $\hat{V}$ are given as the inputs to the illumination process. The reflected ray, $\hat{R}$, is computed according to Snell's law of reflection and geometric optics, which states that the incident angle and the reflected angle are equal for reflections. The reflected vector, $\hat{R}$, can be computed as follows:

$$\hat{R} = 2(\hat{N} \cdot \hat{L})\hat{N} - \hat{L} \tag{3.1}$$

An alternative interpretation of Snell's laws is based on $\hat{H}$, a hypothetical normal that would reflect the light towards the eye. It is computed in the following manner:

$$\hat{H} = \frac{\hat{L} + \hat{V}}{|\hat{L} + \hat{V}|} \tag{3.2}$$

By definition, vectors L, N, and R lie in a common plane, and so do L, H, and V. These two planes, however, do not necessarily coincide.

3.2 Illumination

Illumination models the intensity of light reflected from an object towards the viewer. For a surface point on the object, the illumination computation requires some surface properties (position, normal), some material properties (color, reflectivity, shininess), the viewer's position, and a set of light sources.

Phong's illumination model computes the color of a surface point using the following expression:

$$I_{total,\lambda} = k_a O_\lambda I_{ambient,\lambda} + \sum_{i=1}^{lights} I_{i,\lambda}[k_d O_\lambda (\hat{N} \cdot \hat{L}) + k_s (\hat{V} \cdot \hat{R})^{n_{shiny}}], \qquad (3.3)$$

where I is the light intensity at a particular wavelength denoted by λ; O represents the color of the object; k_a, k_d, and k_s are the reflectance coefficients for ambient, diffuse, and specular light respectively; n_{shiny} is a coefficient of shininess. The other terms represent the vectors described in the previous section: $\hat{N} \cdot \hat{L}$ represents Lambert's diffuse shading, $\hat{V} \cdot \hat{R}$ models specular reflectivity, and n_{shiny} controls the spread of specular highlights.

Blinn's contribution to the illumination model differs only in the computation of specular reflection, where $\hat{N} \cdot \hat{H}$ is used in place of $\hat{V} \cdot \hat{R}$:

$$I_{total,\lambda} = k_a O_\lambda I_{ambient,\lambda} + \sum_{i=1}^{lights} I_{i,\lambda}[k_d O_\lambda (\hat{N} \cdot \hat{L}) + k_s (\hat{N} \cdot \hat{H})^{n_{shiny}}] \quad (3.4)$$

When examining the domains of vectors defining specular highlights (V, R, N, and H), it is generally accepted that Blinn's model is more physically plausible than Phong's. In Blinn's model specular reflection is defined only on the hemisphere above the surface of the object, as opposed to Phong's, where some specular light can be reflected below the surface (Figure 3.2).

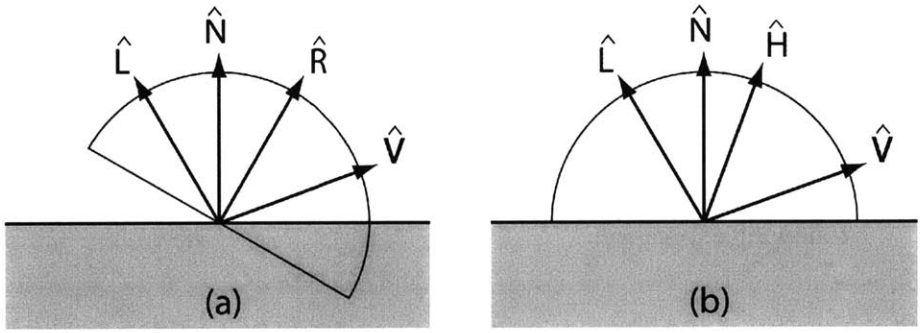

Figure 3.2 Phong (a) and Blinn (b) domain. In (a), the dot product between V and R can be positive, even though V points under the surface of the object. In (b), both N and H point outwards from the surface.

3.3 Texture Mapping

Texture mapping is a powerful technique for adding realism to computer-generated scenes. Fundamentally, texture mapping maps an image, the texture, onto a triangle. The algorithm takes as inputs the texture coordinates of triangle vertices; it then interpolates the texture coordinates linearly in three-space. When this interpolation is computed in 2-D image space, a perspective divide is required to accomplish this interpolation, in order to model the foreshortening typical of perspective.

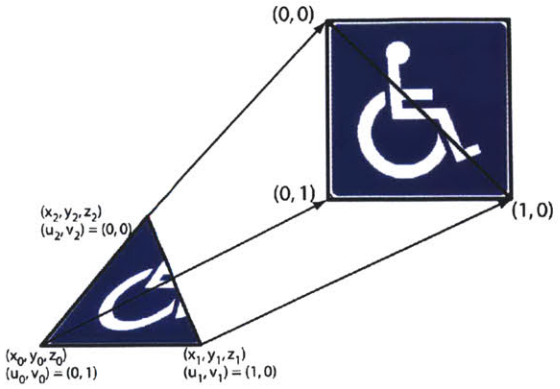

Figure 3.3 Texture mapping. The sign to the right is mapped on the triangle to the left by specifying texture coordinates of the three triangle vertices.

Texture mapping is flexible for two reasons: first, the texture coordinates at triangle vertices can be obtained in many ways; and second, the texture values can be used in many ways. Often, texture coordinates are pre-defined as part of the model description, where they are used to put images onto the object like labels. However, texture coordinates can also be computed using vertex properties and other information; a good example of this is the environment map, where vertex normals are used to index into the texture to simulate global reflection. As for the usage of texture values, they usually stand for the diffuse color of the object. Other uses of texturing include bump mapping, where the texture values represent variations of the surface normal. In this thesis texture values correspond to illumination and are used for shading approximation.

3.4 Parameterization

Depending on the nature of computation, programmers pick different parameterizations of space, i.e. represent the points in the world in terms of

different coordinate systems (origin + basis vectors). When working in screen space, it is convenient to use the classic Cartesian *x-y* coordinate system, where triangle vertices have coordinates (x_0,y_0), (x_1,y_1), (x_2,y_2),. Sometimes, however, it is necessary to view the world relative to the triangle itself. In that case it is common to use a barycentric parameterization. In barycentric coordinate system, *s-t*, the origin is positioned at one of the vertices, and the vectors from the origin to the other two vertices constitute the basis. Thus, coordinates of the triangle vertices are always *(0,0)*, *(1,0)*, and *(0,1)*, which can in some cases greatly simplify the computation.

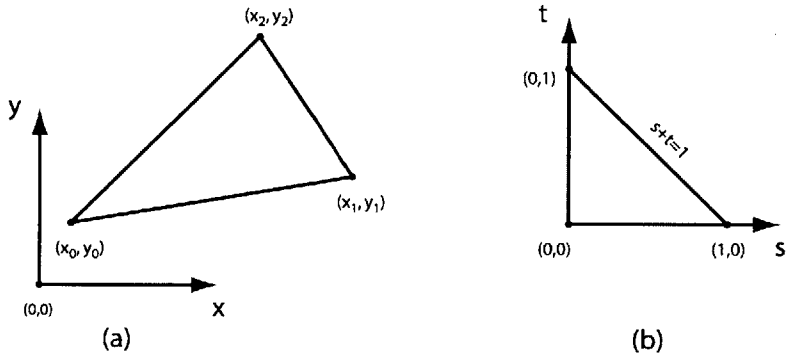

Figure 3.4 Triangle parameterizations. The triangle can be defined by Cartesian coordinates (a), with a global origin at (0,0) and orthogonal basis x and y. They can also be expressed by barycentric coordinates, where the origin is one of the triangle vertices, and the basis vectors are the two edges adjacent to that vertex.

Transitioning from one parameterization to the other is straightforward. The relationship between (x, y) and (s, t) is:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_1 - x_0 & x_2 - x_0 & x_0 \\ y_1 - y_0 & y_2 - y_0 & y_0 \end{bmatrix} \cdot \begin{bmatrix} s \\ t \\ 1 \end{bmatrix} \qquad \{s,t \in [0,1] \mid s + t \le 1\} \qquad (3.5)$$

3.5 Forward Differencing

In a 3D graphics pipeline the triangles are rendered in screen space one scanline at a time. As a result, from one pixel to the next, only one coordinate increases by 1. This property can be exploited to speed up the rendering process by incrementally computing the new value based on the old one, thus reducing the amount of computation needed per pixel. Such technique is called forward differencing.

As an example, I will derive the forward differencing expression for the case of a linear two-dimensional function f:

$$f(x, y) = A \cdot x + B \cdot y + C \tag{3.6}$$

As the renderer traverses a particular scanline, the y coordinate remains constant, and x always increases by one. Hence, the next value of f can be expressed by only one addition to its current value:

$$f(x+1, y) = A \cdot (x+1) + B \cdot y + C = A \cdot x + A + B \cdot y + C = f(x, y) + A \tag{3.7}$$

The same reasoning holds for an increase in y coordinate when transitioning from one scanline to the next.

Forward differencing can be generalized to handle higher-order polynomials by introducing additional state variables. In case of a two-dimensional quadratic function, for example, forward differencing consists of four additions accompanied by three multiplications.

$$\begin{aligned} f(x+1, y) &= A \cdot (x+1)^2 + B \cdot (x+1) \cdot y + C \cdot y^2 + D \cdot (x+1) + E \cdot y + F \\ &= A \cdot x^2 + 2 \cdot A \cdot x + A + B \cdot x \cdot y + B \cdot y + C \cdot y^2 + D \cdot x + D + E \cdot y + F \\ &= f(x, y) + 2 \cdot A \cdot x + A + B \cdot y + D \end{aligned} \tag{3.8}$$

The linear term of equation 3.8 can be computed in separate accumulators as shown below:

$$f(x+1, y) = f(x, y) + g(x, y) \tag{3.9}$$

$$g(x, y) = 2 \cdot A \cdot x + A + B \cdot y + D \tag{3.10}$$

$$g(x+1, y) = g(x, y) + 2 \cdot A \tag{3.11}$$

Since $2 \cdot A$ is constant across each triangle, the amount of computation for forward differencing of a quadratic totals only two additions per pixel. Forward differencing of higher-order functions requires more accumulators.

In this chapter I have presented the mathematical background necessary to understand the illumination and shading stages of the classic graphics rendering pipeline. I have also presented the key insights necessary for understanding the use of texture mapping, barycentric parameterizations of triangles, and forward-differencing interpolation methods. Throughout the reminder of this thesis, I will develop new illumination and shading techniques based on these methods.

Chapter 4

SHADING INTERPOLATION METHODS

In this chapter, I describe Phong shading and its approximations that employ different interpolation strategies to improve rendering quality. The most basic of these is the widely used linear Gouraud shading. Additionally, I will show how to quadratically shade triangles and achieve higher rendering quality with a reasonable additional cost compared to Gouraud.

4.1 Phong Shading

Phong shading, as described in Chapter 2, was introduced by Phong as the appropriate shading method to go along with his illumination model. Triangles are shaded by linearly interpolating the surface normal, defined at each vertex, and re-evaluating the illumination equation at each pixel. Qualitatively, images generated using Phong shading are superior to those generated by other methods in the standard rendering pipeline. However, Phong shading is computationally demanding and it does not run in real-time even on modern hardware.

Phong shading is expensive not only because illumination has to be evaluated at each pixel, but also because linearly interpolating the normal is not as simple as linearly interpolating a scalar. Most commonly, the normal is interpolated by linearly interpolating each of its scalar components and re-normalizing at each pixel. Normalization includes a divide, a square root, and a few multiplications and additions. To avoid re-normalizing the normal, some researchers have considered quadratically interpolating its components. This approach works

reasonably well when objects are some distance away from the eye, but the quality decreases as they come closer and re-normalization becomes necessary. Another approach is to spherically interpolate the normals, i.e. map the triangle onto the surface of a sphere and use the sphere normals as the triangle normals. Although spherical interpolation works well, it still requires a significant amount of computation per pixel comprising of multiplications, additions and trigonometric functions.

4.2 Linear Gouraud Shading

One of the first and well-known approaches to speeding up shading while keeping the surfaces smooth is Gouraud shading [1]. Gouraud applies the illumination only at triangle vertices, and linearly interpolates the resulting colors across the triangle. Linear interpolation, however, cannot express the light intensity peaks, or highlights, at regions of high curvature. Gouraud shading has a common artifact that it is easy to recognize the underlying triangular mesh, particularly for low-resolution meshes. This results from the fact that the derivative of the piece-wise linear color function is not continuous across the triangle edges. These discontinuities are accentuated by the human visual system, through a psychophysical Phenomenon know as Mach banding [11].

4.3 Quadratic Shading

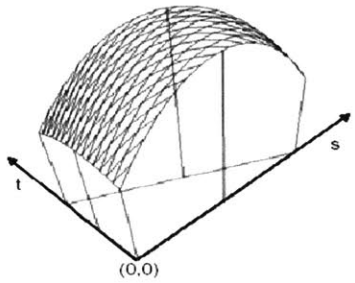

Figure 4.1 Quadratic Interpolation. Quadratic color interpolation can be defined over the surface of a triangle to better approximate Phong shading. (figure used by permission of [5])

An alternative to linear Gouraud shading is quadratic interpolation. A quadratic can express shading maximums and minimums that appear within a triangle's interior, and thus can approximate light distribution better than Gouraud shading. In addition, quadratic shading can be constrained to enforce continuity across triangle edges, thus masking the underlying mesh. The downside of this approach is an increase in computation. Since a quadratic has more degrees of freedom (six), we need more constraints to completely define it. Therefore, illumination must be computed at six points, as opposed to three. Furthermore, evaluating the quadratic terms during scan conversion introduces additional per-pixel overhead compared to the linear Gouraud shading, as mentioned in section 3.5.

Although more complex than Gouraud, setting up a quadratic across a triangle is straightforward. For every pixel with barycentric coordinates (s, t) within the triangle, we can define the color value r(s, t) according to the following equation:

$$r(s,t) = C_0 + C_1 \cdot s + C_2 \cdot t + C_3 \cdot s^2 + C_4 \cdot s \cdot t + C_5 \cdot t^2, \qquad (4.1)$$

where C_0-C_5 are constant coefficients per triangle. To solve for these coefficients, we need six pre-computed color values. Three of the values are taken at triangle vertices, which already have their positions and normals computed. To enforce color continuity across triangles, the remaining three values must be located on the triangle edges – one per edge. This way the three values per edge – two at the vertices and one in between – completely define a quadratic (Ax^2+Bx+C). Even with abovementioned constraints, edge points can be located anywhere between the vertices, and one could formulate picking them as an optimization problem. However, for simplicity of computation, the edge points are preset at the midpoints in-between the vertices, as illustrated in figure 4.2.

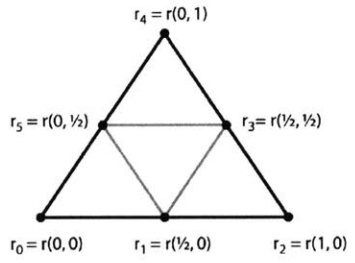

Figure 4.2 Evaluated points for quadratic shading. Six points at the vertices and on the edges are picked for cross-triangle continuity.

Once we have the six color values, we can set up a linear system of six equations. As described in [5], the system yields the solution:

$$\begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \end{bmatrix} = \begin{bmatrix} r_0 \\ -3\cdot r_0 + 4\cdot r_1 - r_2 \\ -3\cdot r_0 - r_4 + 4\cdot r_5 \\ 2\cdot r_0 - 4\cdot r_1 + 2\cdot r_2 \\ 4\cdot r_0 - 4\cdot r_1 + 4\cdot r_3 - 4\cdot r_5 \\ 2\cdot r_0 + 2\cdot r_4 - 4\cdot r_5 \end{bmatrix} \tag{4.2}$$

Finally, the following procedure can be used to quadratically shade a triangle:

1. Evaluate the illumination equation at the three vertices and three midpoints. Surface normal at some midpoint is the bisector of two neighboring vertex normals – it is computed by normalizing their sum.
2. Compute the coefficients C_0-C_5 using equation 4.2.
3. For each pixel, evaluate the quadratic equation $r(s, t)$, equation 4.1, to get the output color. Knowing the barycentric coordinates (s, t) for each pixel is simple: both coordinates should be linearly interpolated across the triangle (with the perspective divide), exactly the way texture mapping does it. Their values at triangle vertices are $(u_0, v_0) = (0,0)$, $(u_1, v_1) = (1,0)$, and $(u_2, v_2) = (0,1)$. These facts can be derived directly from equation 3.5, yielding:

$$\begin{bmatrix} s \\ t \end{bmatrix} = \frac{\begin{bmatrix} y_0 - y_2 & x_2 - x_0 & x_0 \cdot y_2 - x_2 \cdot y_0 \\ y_1 - y_0 & x_0 - x_1 & x_1 \cdot y_0 - x_0 \cdot y_1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}}{(x_2 - x_0)\cdot(y_1 - y_0) - (x_1 - x_0)\cdot(y_2 - y_0)} \tag{4.3}$$

The described algorithm can be generalized to perform higher order shading. A cubically shaded triangle, for example, requires ten lighting evaluations It is equivalent to illuminating nine smaller Gouraud shaded triangles with more expensive per-pixel computation (Figure 4.3). In view of that, it is not clear whether the rendering quality of one larger cubically shaded triangle is noticeably

better than that of nine smaller linearly shaded triangles. Higher dimensional shading introduces even more complexity and one should rather subdivide the triangle and use simpler color interpolation on several smaller triangles.

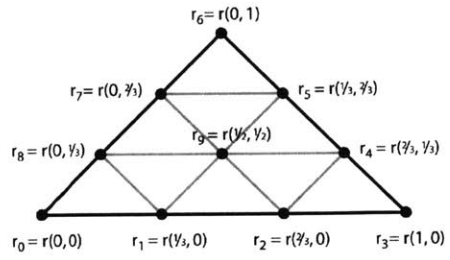

Figure 4.3 Evaluated points for cubic shading. Ten points are necessary to define a cubic over the surface of a triangle. Nine of them have to be on the edges to enforce continuity; the last one is usually picked in the triangle center.

In this chapter I have concentrated on shading methods that use different interpolation techniques. I described Phong shading, the high-quality shading method, which interpolates the surface normal and illuminates at each pixel, but does not run at interactive rates. I have also described Gouraud shading, the method commonly used by current realtime 3D graphics hardware, which illuminates at triangle vertices and linearly interpolates the color. Finally, I have presented quadratic shading, which illuminates at six points per triangle and quadratically interpolates the color. Quadratic shading has higher quality than the linear, plus it can be implemented in hardware. In the next chapter I will introduce shading methods that utilize texture mapping, after which I will proceed to analysis.

Chapter 5

TEXTURING APPROACHES

Texture mapping, as described in section 3.3, is a very flexible and powerful rendering tool. Since efficient texturing is now commonly available in hardware, texturing approaches have been developed to implement a myriad visual effects, including shading. There are many ways in which textures can be used for simulating Phong shading. These various methods differ in the way texture coordinates are computed the triangle vertices, as well as which textures are used. Since Gouraud shading works reasonably well for diffuse reflection, texture mapping is most useful for emulating specular highlights. In the implementations described in this thesis, all texturing approaches use Gouraud shading for diffuse reflection, and textures only for specularities.

Texturing methods described in this chapter include environment mapping, a very common technique today, my own method called 'Blinn mapping', and an extension to it called 'quadratic Blinn mapping'.

5.1 Environment Map

Environment mapping is a commonly used texture mapping technique for rendering highly reflective and specular materials. Conceptually, environment mapping wraps an image of the environment around the object. Therefore, simulating specular highlights is equivalent to rendering an environment composed of extended light sources. Modern 3D graphics API's (Direct3D, OpenGL) offer several variations of environment mapping: spherical, bi-

quadratic, longitude, and cube mapping [8, 9]. However, implementations vary from one library to another. For example, in DirectX documentation [9] spherical map texture coordinates are described as view-independent, while in OpenGL [8] they are view-dependent. In order to be physically valid, reflectance mapping has to be view-dependent. This stems from the fact that intensity of specular reflection is computed using the view vector (Section 3.2). For this reason, I will not consider view-independent approximations. Of the view-dependent implementations, I will focus my discussion on spherical mapping, because it is easier to analyze. The analysis, however, applies to the other view-dependent environment mapping methods as well.

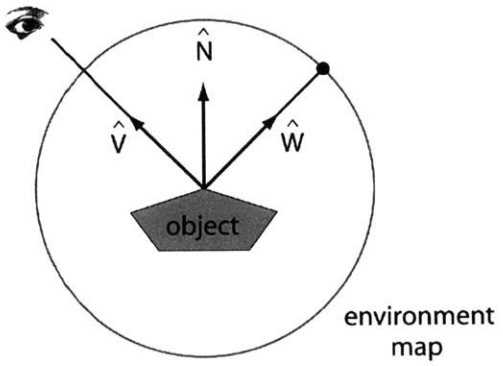

Figure 5.1 Environment mapping. In all types of environment mapping, the environment is assumed to be at infinity and is accessed by the reflected view vector W.

All view-dependent environment mappings cast the reflected view vector, $\hat{W}$ (not to be confused with reflected light, $\hat{R}$, from section 3.1), into the environment. Note that the light direction plays no role in texture coordinate computation. Spherical mapping computes texture coordinates (u, v) as follows:

$$m = \sqrt{W_x^{\,2} + W_y^{\,2} + (W_z + 1)^2} \tag{5.1}$$

$$u = \frac{1}{2} + \frac{1}{2}\frac{W_x}{m} \tag{5.2}$$

$$v = \frac{1}{2} + \frac{1}{2}\frac{W_y}{m} \tag{5.3}$$

The texture used for spherical mapping is depicted in Figure 5.2; specular highlights can be drawn at correct locations in the environment. The details of creating the texture are not relevant for this discussion.

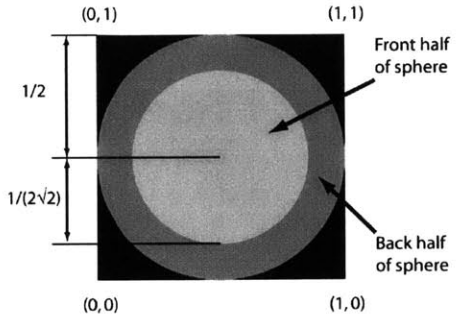

Figure 5.2 Spherical mapping texture. Both the front and the back of the environment sphere are mapped onto the rectangular texture

As will be discussed in the next chapter, environment mapping cannot simulate point lights, nor can it correctly approximate Phong shading with Blinn's illumination model.

5.2 Blinn Map

Another texturing method for shading is described as "Phong map" in [7]. It uses the projection of light vector onto the plane orthogonal to the vertex normal to index into the texture. Clearly, this is only valid for diffuse reflection, since it

permits no view dependence, but it can easily be extended to support specularity. With slight modifications, I propose projecting vertex normal onto the plane defined by light direction for diffuse lighting. I extend this method to support specular reflections by projecting the normal onto the plane defined by the half-vector, H. The two texture values are combined for the final result. I call this method "Blinn map" since it is based on Blinn's illumination equation (Figure 5.3).

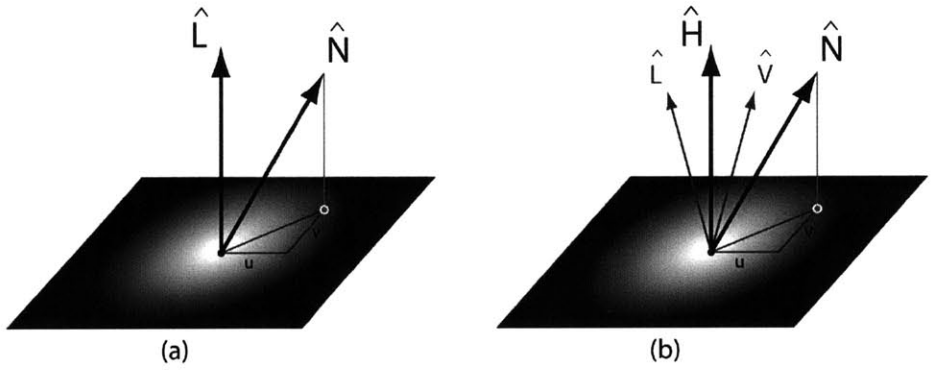

Figure 5.3 Blinn map for diffuse (a) and specular (b) reflections. In (a), the surface normal is projected onto the plane defined by the light vector for diffuse shading. In (b), the plane is defined by the half-vector for specular shading.

Blinn map, in contrast to environment maps, incorporates light direction into the computation of texture coordinates, thus enabling it to simulate accurate point and directional lights.

The appropriate textures for simulating Phong shading with Blinn maps are essentially lookup tables of dot products that approximate diffuse and specular terms of the illumination equation (3.4). For diffuse shading, the texture is simply an array of dot products that approximate $\hat{N} \cdot \hat{L}$. For specular highlights, the dot products are raised to the power of n_{shiny} in order to approximate $(\hat{N} \cdot \hat{H})^{n_{shiny}}$.

The exact computation takes into account the fact that Blinn mapping projects vectors onto planes to get texture coordinates. Therefore, the diffuse texture value at point $p(x, y)$ in the texture plane is the value of the dot product of two unit-vectors $v_P \cdot v_N$, as shown in Figure 5.4. The vector v_P is kept constant at $(0,0,1)$ – it defines a shading parameter plane that is perpendicular to it; v_P represents the light direction, $\hat{L}$, for diffuse, and the half-vector, $\hat{H}$, for specular texture. The other vector, v_N, also originates from the center of the texture with its projection onto the shading parameter plane being (x, y). It represents the vertex normal N for both diffuse and specular texture. The specular texture uses the same dot product, but exponentiates it to the power of n_{shiny}.

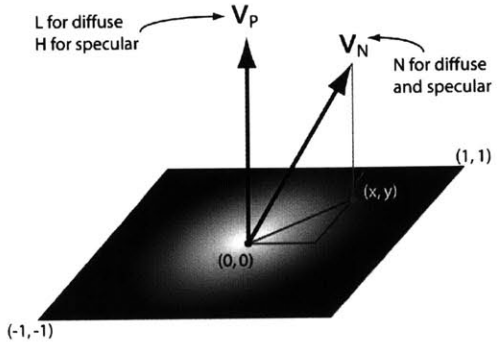

Figure 5.4 Blinn texture.

$$diffuse(x, y) = (0,0,1) \cdot (x, y, \sqrt{1 - x^2 - y^2}) = \sqrt{1 - x^2 - y^2} \tag{5.4}$$

$$specular(x, y) = (1 - x^2 - y^2)^{n_{shiny}/2} \tag{5.5}$$

Another dilemma when shading with Blinn maps is how to compute texture coordinates from the projected vectors. Basically, the texture coordinates are

computed by projecting the normal, v_N, onto the shading parameter plane and assigning a two-dimensional coordinate. Obviously, the resulting coordinates depend on the orientation of the coordinate system in the shading parameter plane. Unfortunately, the choice of basis axes for that coordinate system directly affects the rendering quality; thus, some care has to be taken in choosing it.

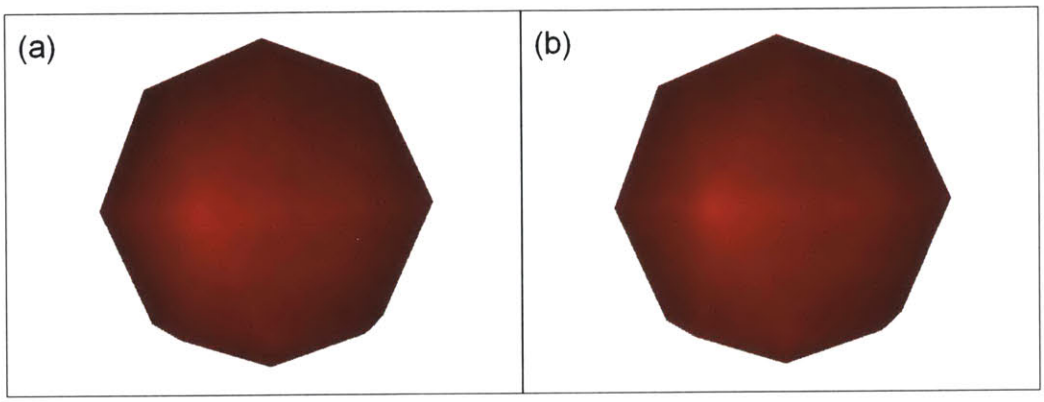

Figure 5.5 Worst case Blinn axis. If one of the basis axis always aligns to the projected vector, the output of Blinn mapping (b) is comparable to Gouraud shading (a).

In a general 3D scene, shading parameter planes differ at each vertex, and every set of texture coordinates is computed in a unique coordinate system. If the bases for those coordinate systems are assigned independently of each other, the rendering quality can significantly vary. In the worst case, one of the axes is aligned with the projected vector, resulting in v-coordinate always being zero. Hence the whole image is rendered from only one line in the texture, and the output is comparable to that of Gouraud shading (Figure 5.5). In practice, scenes look best if one of the basis axis is chosen as some global vector (global axis), or more precisely the re-normalized projection of some global vector onto the shading parameter plane. The other axis can be obtained by computing the cross product of the plane normal with the first global axis. This way the axes are correlated from triangle to triangle and edge transitions blend well. Still,

rendering quality drops off around points where the global axis aligns with shading parameter plane normal – this results in a singularity, as the global axis projects to a point, and the basis vectors have no length (Figure 5.6). Therefore, a good choice for the global axis is the up vector of the camera coordinate system – this way anomalies will be shifted away from the view.

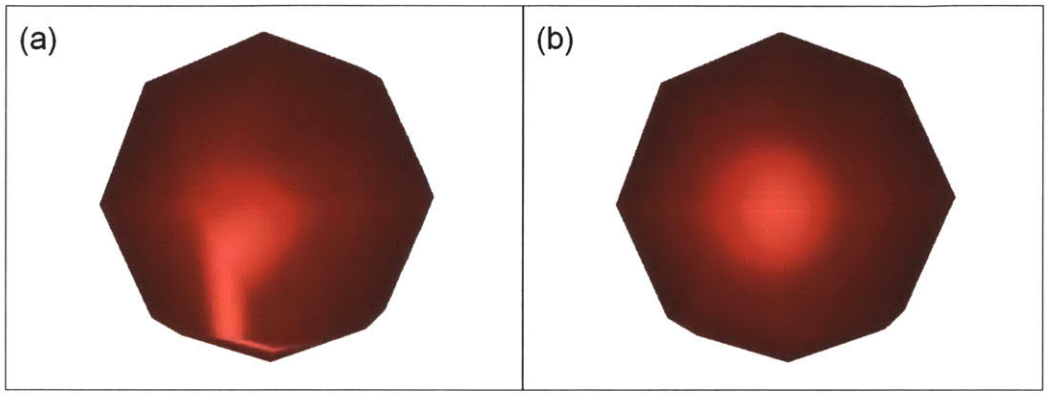

Figure 5.6 Blinn axis singularity. Anomalies appear around the point where global axis is aligned with the shading parameter plane normal (a). They can be concealed by shifting the axis away from the view vector (b).

In summary, Blinn map computes diffuse and specular texture coordinates, (u_D,v_D) and (u_S,v_S) respectively, as shown in Figure 5.5 and the subsequent equations. Here, $\hat{u}$ is the re-normalized projection of the camera's up vector onto the shading parameter plane, and $\hat{v}$ is a vector perpendicular to it.

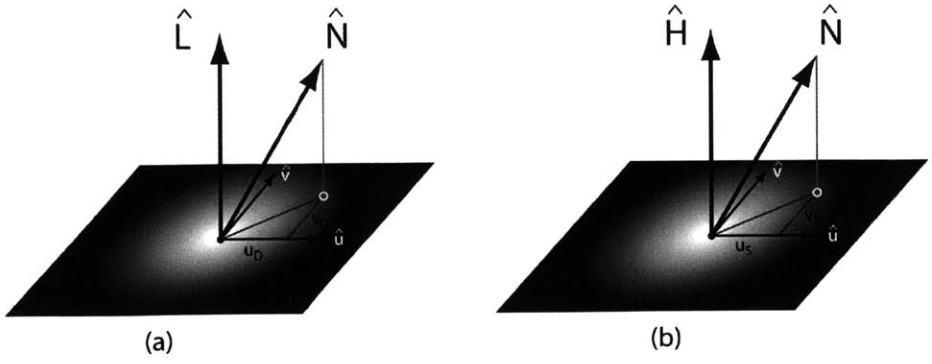

Figure 5.7 Blinn texture coordinates. After projecting the specified vectors onto the shading parameter plane, a set of coordinates is assigned according to the u-v coordinate system.

$$u_D = \hat{u} \cdot (\hat{N} - (\hat{N} \cdot \hat{L})\hat{L})$$
$$v_D = (\hat{L} \times \hat{u}) \cdot (\hat{N} - (\hat{N} \cdot \hat{L})\hat{L}) \tag{5.6}$$

$$u_S = \hat{u} \cdot (\hat{N} - (\hat{N} \cdot \hat{H})\hat{H})$$
$$v_S = (\hat{H} \times \hat{u}) \cdot (\hat{N} - (\hat{N} \cdot \hat{H})\hat{H}) \tag{5.7}$$

5.3 Quadratic Interpolation of Texture Coordinates

All of the previously mentioned texturing approaches compute texture coordinates at triangle vertices, and let the texturing hardware take care of interpolation and rendering. Nearly all existing graphics hardware is designed to linearly interpolate texture coordinates (with the perspective divide) across the triangles. However, a better texture mapping, resulting in a closer Phong shading approximation, can be achieved by quadratically interpolating the texture coordinates. The setup for this computation follows from the quadratic shading in section 4.3 – evaluate texture coordinates at the six points around the triangle and use the quadratic to interpolate them for the remaining pixels.

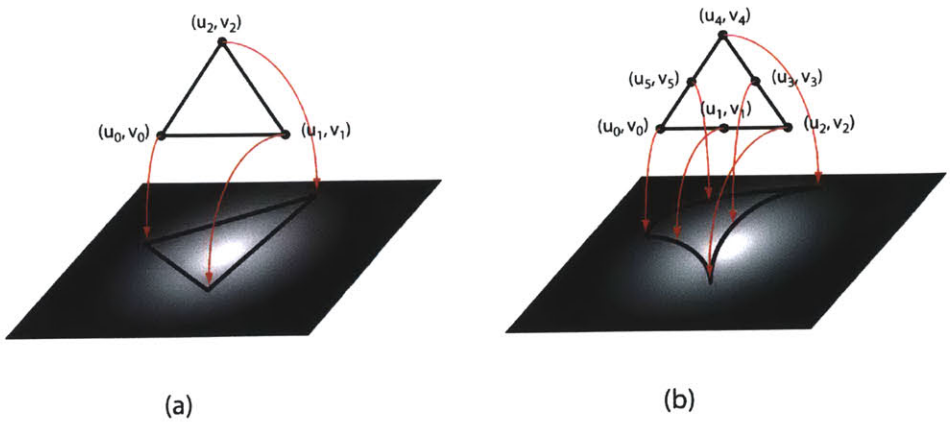

Figure 5.8 Quadratic Blinn mapping. Regular Blinn map (a) evaluates texture coordinates at three points and linearly interpolates them. Quadratic Blinn map (b) evaluates the coordinates at six points and quadratically interpolates them.

In this thesis, quadratic texture coordinate interpolation is combined with the Blinn map to improve rendering quality; hence, it is referred to as "quadratic Blinn map".

In this chapter I have presented three methods that utilize texture mapping to approximate Phong shading: environment mapping, Blinn mapping, and quadratic Blinn mapping. In the remainder of the thesis I will analyze all the methods introduced in this and the previous chapter.

Chapter 6

RESULTS AND ANALYSIS

This chapter presents the results, i.e. rendered scenes, of the described Phong shading approximation techniques: quadratic interpolation, environment map, Blinn map, and quadratic Blinn map (Gouraud shading has already been analyzed and evaluated by various researchers). The outputs are compared to the output of Phong shading with Blinn's illumination model and linear interpolation of normals (see section 4.2), which is considered the standard in this thesis. As mentioned in Chapter 5, the texturing approaches use textures only for specular highlights, thus only the specular components of the outputs will be compared. Furthermore, the proposed approximation methods are examined analytically to assess their accuracy compared to Phong shading. Finally, their efficiency is considered. Since the implementation is done in Java without hardware acceleration, relative speeds rather than absolute performance should be considered.

All the analysis is done on two models shown in Figure 6.1, as rendered by Phong shading. First is the Utah teapot model with 3750 facets and a white directional light, a representative of higher-resolution meshes. The second model is a single blue triangle with a white point light in front of it, designed to examine rendering of low-resolution meshes and point lights. For comparison, Figure 6.2 shows the meshes rendered by Gouraud shading, the way current hardware would render them.

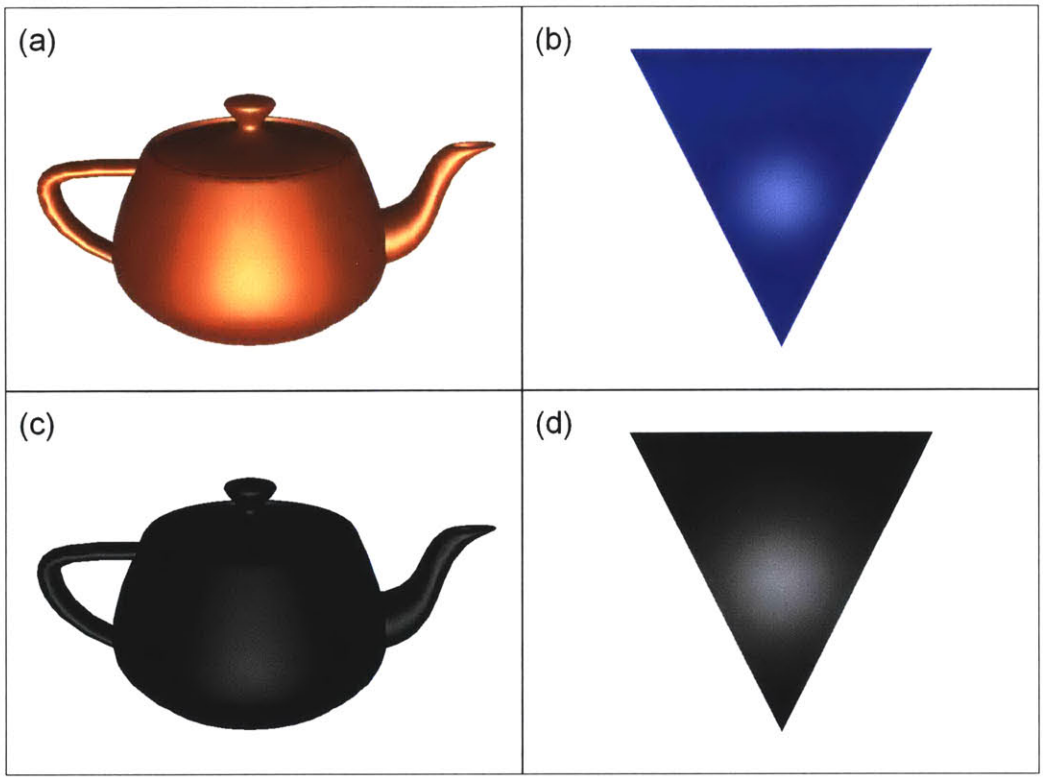

Figure 6.1 Phong shaded teapot (a) and triangle (b), followed by corresponding renderings with only specular reflection in (c) and (d). These images are the reference for the analysis of fake Phong shading methods.

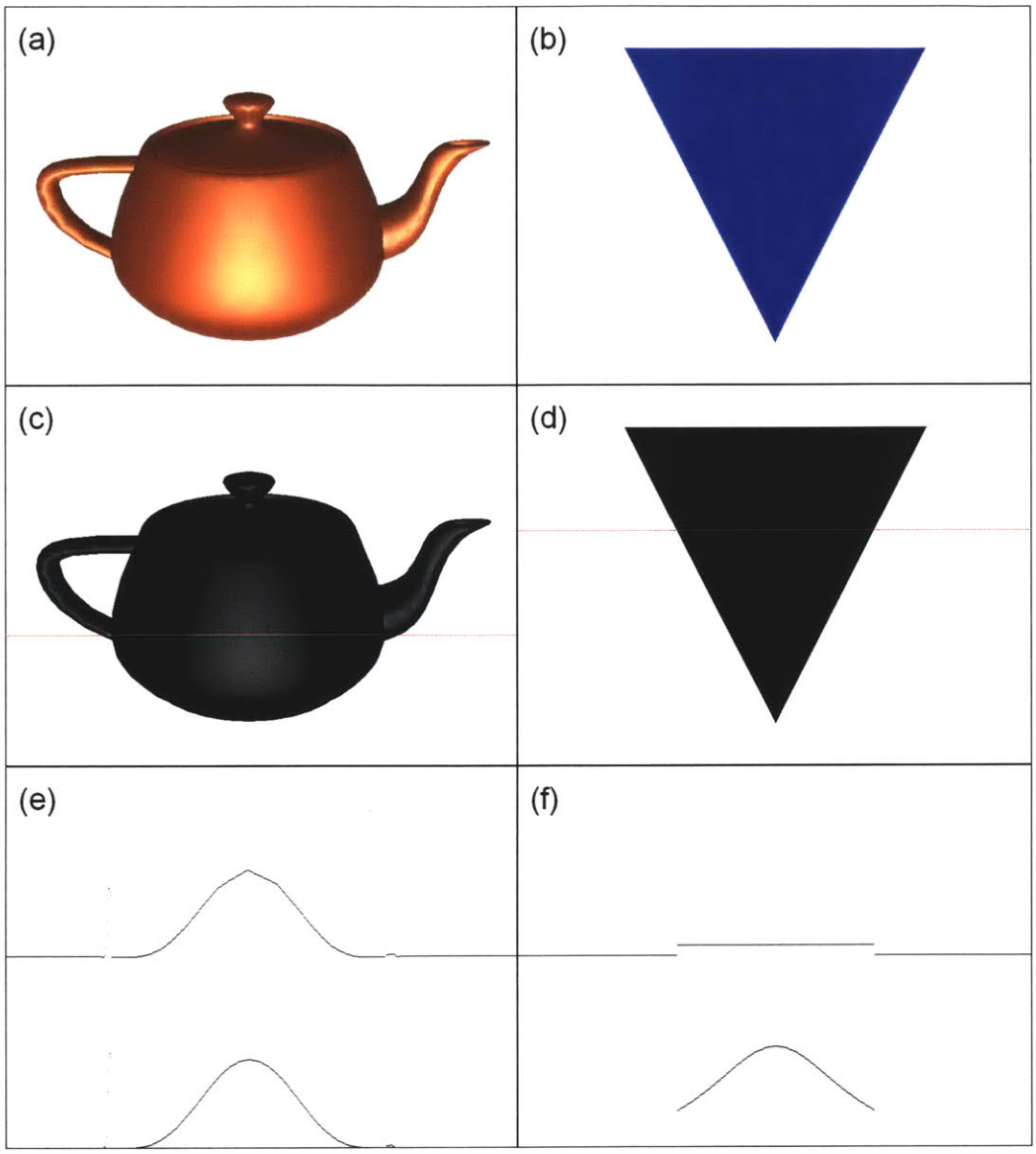

Figure 6.2 Gouraud shaded teapot (a) and triangle (b), followed by corresponding renderings with only specular reflection in (c) and (d), as well as the scanline illumination curves in (e) and (f). Gouraud shading is how current hardware renders triangular meshes.

6.1 Comparison Methods

The evaluated algorithms (Phong shading approximations) are compared to the standard (Phong shading) using these few comparison methods:

1. Color Correctness

   The evaluated output image is compared to the standard Phong output image on pixel-by-pixel basis. The resulting metric indicates how close the colors of the two images are. Specifically, color correctness is a number in the range [0, 1], representing one minus the normalized average color difference. The closer this number is to 1, the more alike are the compared images. The exact formula for color correctness is:

$$c = 1 - \frac{\sum_{i=1}^{N} (|R_i - r_i| + |G_i - g_i| + |B_i - b_i|)}{3 \cdot 255 \cdot N} \tag{6.1}$$

   Here c is the color correctness; N is the number of non-background pixels in the images; R_i, G_i, and B_i are red, green, and blue components of a non-background pixel i in the standard image; r_i, g_i, and b_i are the corresponding color components in the evaluated image.

   Color correctness values for all Phong approximation methods against Phong shading are shown in Table 6.1. Because of the formula used for computing them (Equation 6.1), the numbers are relatively densely distributed, thus even small differences are significant. Furthermore, color correctness represents average color deviation between two images, revealing nothing about characteristic

features such as highlights. Those aspects are better exposed by the next comparison method – the scanline similarity.

In addition to Table 6.1, color correctness is shown as a difference image presented with each fake Phong shading method. Such images represent the color difference between the renderings of Phong shading and a particular approximation method. Furthermore, difference images are are gamma-enhanced, since colors are usually very close. The formula for computing a pixel of the difference image is:

$$[r,g,b]=255\cdot\left[\left(\frac{|R_i-r_i|}{255}\right)^{1/10},\left(\frac{|G_i-g_i|}{255}\right)^{1/10},\left(\frac{|B_i-b_i|}{255}\right)^{1/10}\right], \qquad (6.2)$$

where r, g, and b are the output color components of pixel i; R_i, G_i, and B_i are the corresponding color components in the standard image; finally, r_i, g_i, and b_i are the colors of the same pixel in the evaluated image.

2. Scanline Similarity

This method computes illumination values across the same scanline in both evaluated and standard output image, resulting in two graphs. For simplicity, the scanline is located approximately halfway down the image and includes the most interesting highlights. Subsequently, the resulting graphs can be visually analyzed comparing values, peak locations and intensities, as well as the smoothness of the curves.

The illumination value of a pixel falls between 0 and 1, and is computed as follows:

$$I(x,y) = \frac{R(x,y) + G(x,y) + B(x,y)}{3*255}, \tag{6.3}$$

where R, G, and B are color components of a pixel whose coordinates are x and y.

3. Analytic Analysis

This method applies primarily to texturing approaches. It analytically computes whether colors at vertices are exact, i.e. whether texture coordinate computation and texture generation are consistent with the illumination equation. This method also indicates how much the interpolated values deviate from the standard.

4. Efficiency Evaluation

Due to the nature of my implementation, efficiency and speed cannot be objectively assessed. Therefore, for each approach I will evaluate the complexity of necessary initialization and per-pixel computation, and then relatively rank all approaches. In addition, I will mention any implications to using hardware for real-time implementations of the analyzed fake Phong shading methods.

Table 6.1 Color correctness values for Phong shading approximations. Gouraud shading with subdivision shades a mesh where each triangle is subdivided into four smaller triangles, such that evaluation points are the same as in quadratic shading (see Figure 4.2).

Model	Gouraud Shading	Gouraud with subdivision	Quadratic Shading	Environment Map	Blinn Map	Quadratic Blinn Map
Teapot	0.9870	0.9939	0.9949	0.9872	0.9886	0.9888
Triangle	0.7644	0.8970	0.9058	0.5589	0.8205	0.9311

6.2 Quadratic Shading

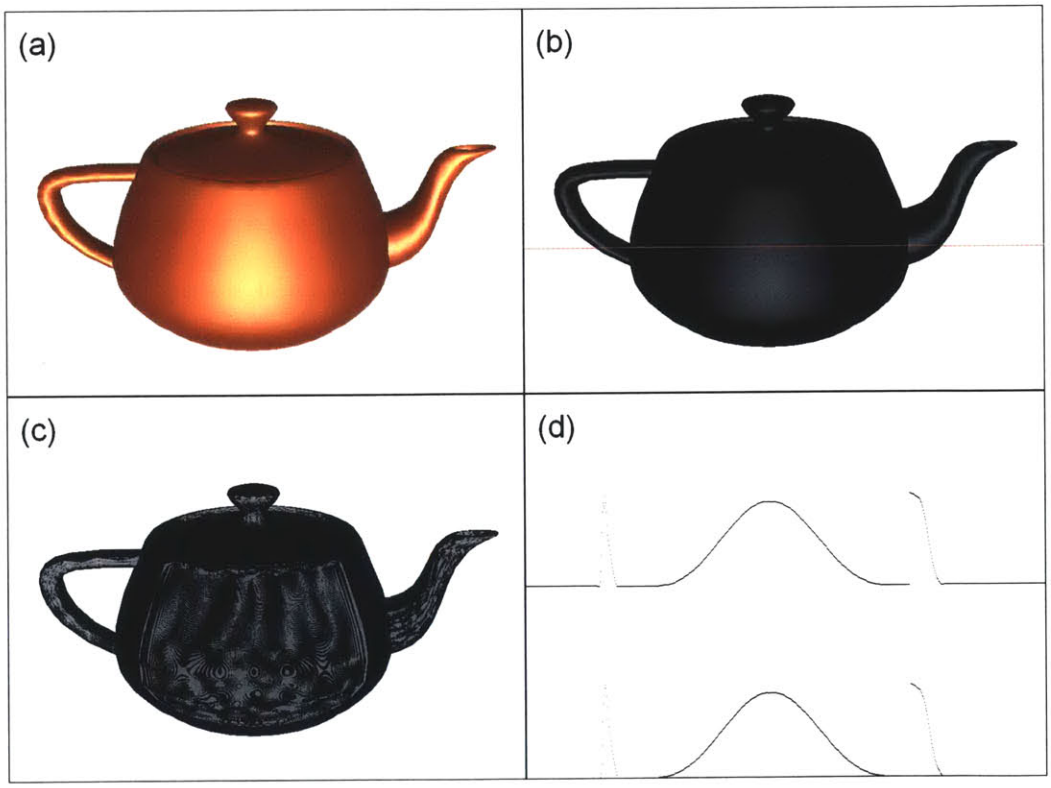

Figure 6.3 Teapot rendered by quadratic shading. The full rendering is shown in (a), and the specular component along with the analyzed scanline in (b). The difference image is in (c), and the scanline illumination curves are in (d). The lower curve belongs to Phong shading.

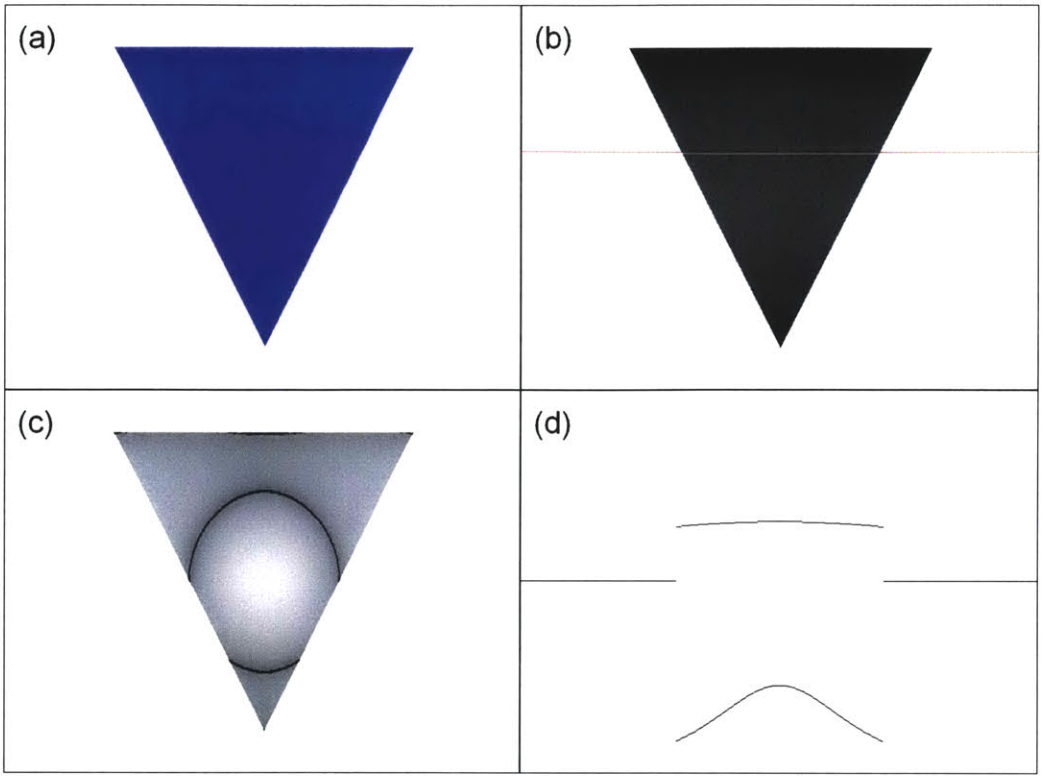

Figure 6.4 Triangle rendered by quadratic shading. The full rendering is shown in (a), and the specular component along with the analyzed scanline in (b). The difference image is in (c), and the scanline illumination curves are in (d). The lower curve belongs to Phong shading.

From the rendered image and the scanline illumination graph of the teapot (Figure 6.3), it is obvious that quadratic interpolation handles relatively fine meshes rather well. The illumination curve is smooth and closely follows peeks of Phong shading. The triangle model (Figure 6.4), however, reveals the weakness of quadratic shading – rendering quality drops severely on low level meshes. The illumination curve shows a peek, but its intensity is not nearly as high as it should be. This, of course, is due to the fact that samples for quadratic shading are taken from the vertices and edge midpoints, letting the quadratic equation approximate shading at all other points. Therefore, only the original six

points will have correct shading values. The colors of remaining pixels come very close to Phong shading provided that view and light directions do not change much across the surface of the triangles – but those are the same cases where linear shading performs well.

Comparing figures 6.2, 6.3, and 6.4, it is obvious that the quality of quadratic shading is higher than that of Gouraud shading. This comes as no surprise, since quadratic shading evaluates the illumination equation at more points per triangle. Therefore, quadratic shading should actually be compared to Gouraud shading of a subdivided mesh, where each quadratically shaded triangle corresponds to four linearly shaded subtriangles, as illustrated in Figure 4.2. Teapot and triangle renderings using subdivided Gouraud shading are presented in figures 6.5 and 6.6. Those two figures, as well as the color correctness values from Table 6.1, indicate that quadratic shading is more accurate than even subdivided Gouraud shading. This is not obvious from renderings of high-resolution meshes, but is clearly visible on coarser meshes.

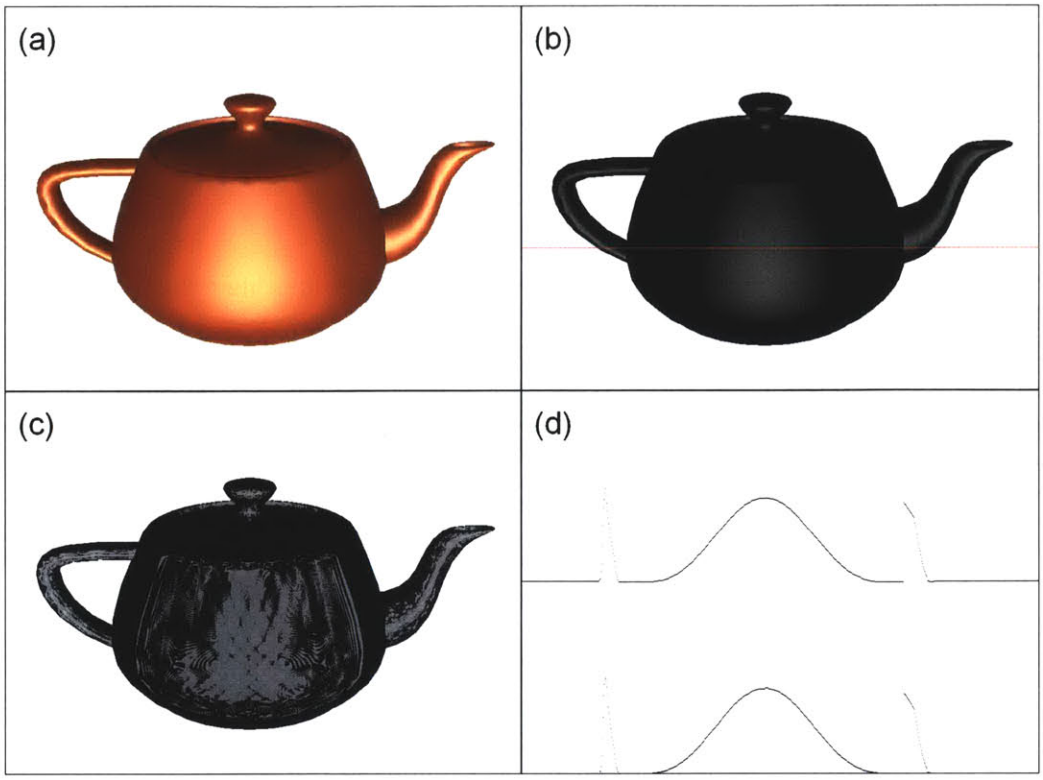

Figure 6.5 Teapot rendered by subdivided Gouraud shading. The full rendering is shown in (a), and the specular component along with the analyzed scanline in (b). The difference image is in (c), and the scanline illumination curves are in (d). The lower curve belongs to Phong shading.

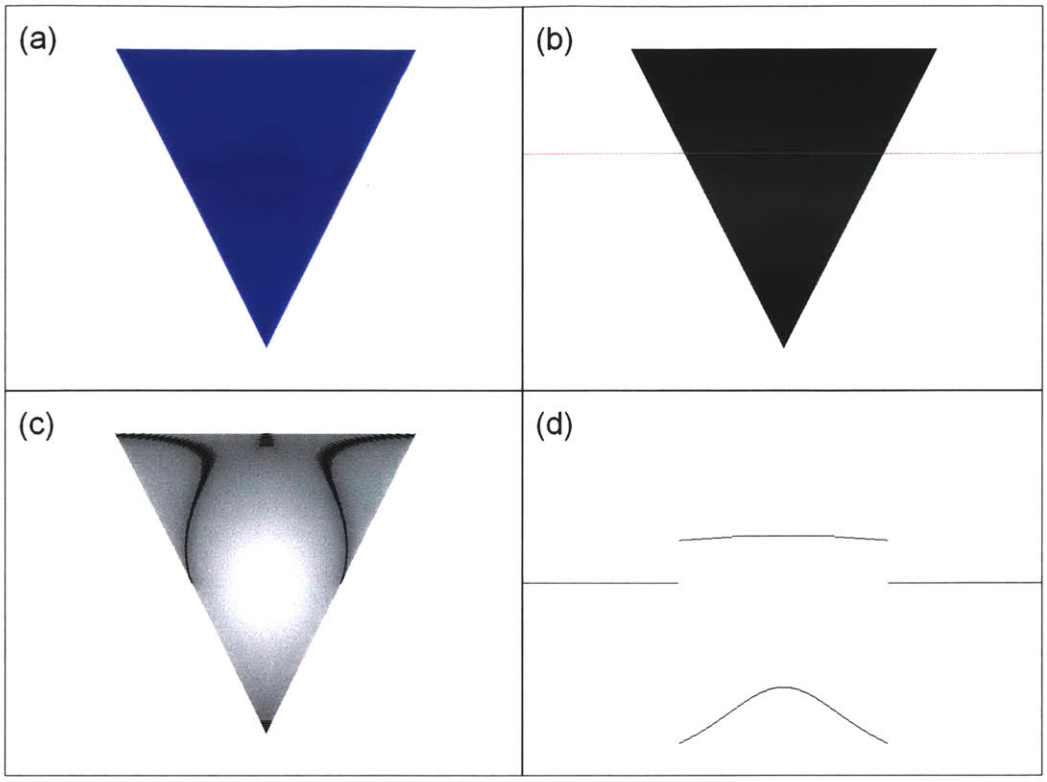

Figure 6.6 Triangle rendered by subdivided Gouraud shading. The full rendering is shown in (a), and the specular component along with the analyzed scanline in (b). The difference image is in (c), and the scanline illumination curves are in (d). The lower curve belongs to Phong shading.

In terms of computation costs, quadratic shading as described in this thesis requires a setup overhead of several multiplications and additions in order to compute the quadratic coefficients. However, the quadratic equation has to be recomputed at each pixel – forward differencing cannot be applied because the barycentric coordinates (s, t) do not increase by 1 when screen coordinates (x, y) do. Thus, quadratic shading performs significantly slower than Gouraud shading – in my Java implementation it is two times slower. Nevertheless, with current hardware vertex (v.1.1) and pixel (v.1.4) shaders it is possible to implement a real-time quadratic shader supporting one light source. At only a small cost to frame-

rate, the improved quality of hardware-implemented quadratic shading makes it more attractive for 3D applications that currently use linear shading.

To sum up, quadratic shading, though more complicated than Gouraud, runs faster than Phong shading and can be implemented in hardware. It handles higher-resolution meshes well, improving rendering quality over linear shading and better masking the triangle boundaries. With lower-resolution meshes, however, rendering quality of quadratic shading becomes inadequate, leaving Gouraud shading more convenient in such cases.

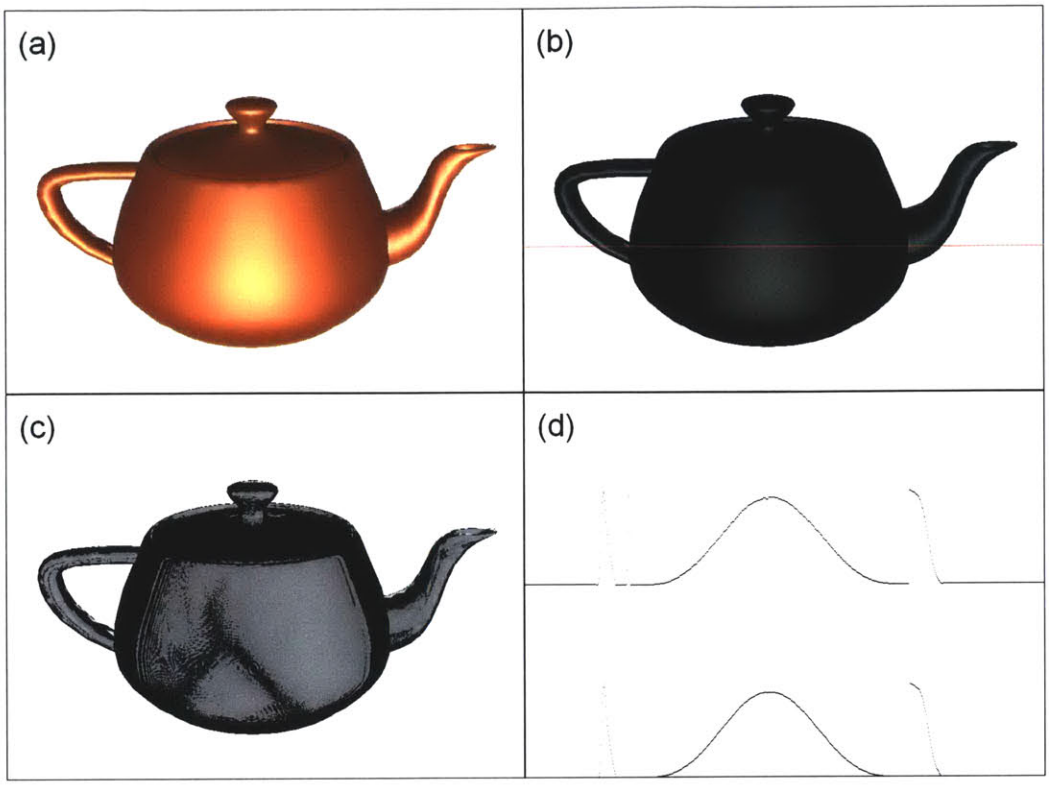

Figure 6.7 Teapot rendered by enviroment map. The full rendering is shown in (a), and the specular component along with the analyzed scanline in (b). The difference image is in (c), and the scanline illumination curves are in (d). The lower curve belongs to Phong shading.

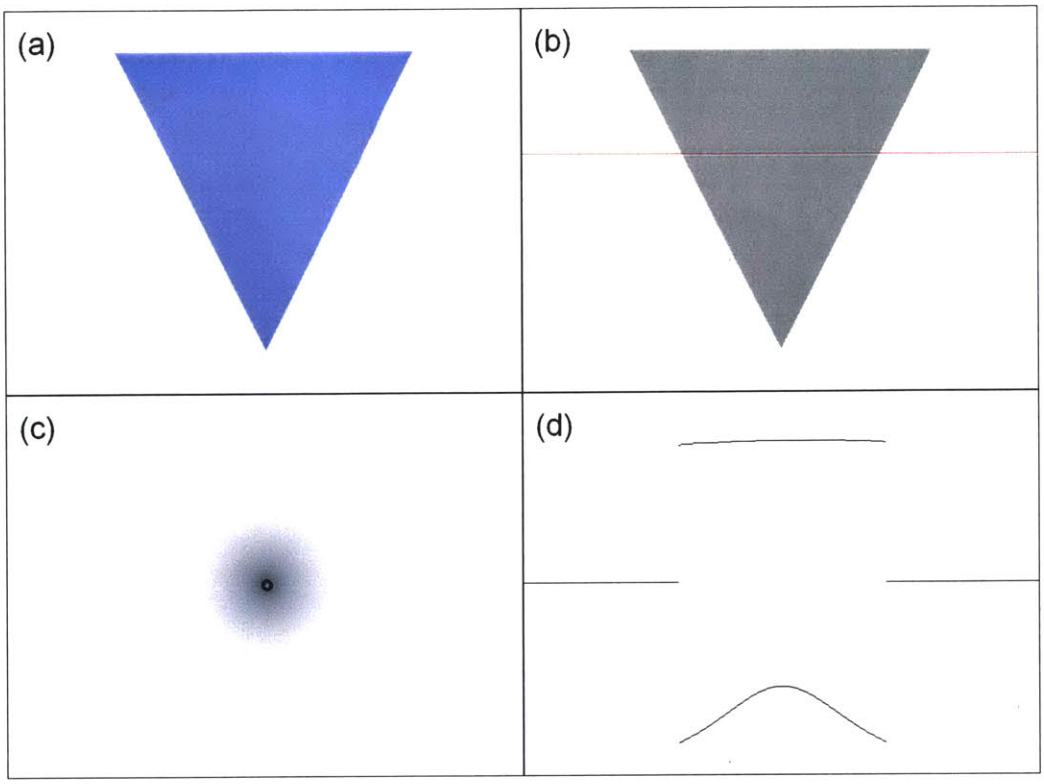

Figure 6.8 Triangle rendered by environment map. The full rendering is shown in (a), and the specular component along with the analyzed scanline in (b). The difference image is in (c), and the scanline illumination curves are in (d). The lower curve belongs to Phong shading.

The teapot rendering reveals that environment mapping does well with fine meshes – illumination curve is smooth and peeks are comparable to those of Phong shading. However, some color distortions are visible near the edges of the handle and the spout: those are due to sampling artifacts of the sphere map and can be alleviated using other environment mapping techniques [8]. The triangle rendering, which contains a point light, is a poor approximation of Phong shading: the specular highlight does not even exist, and the illumination graph shows no peeks. Analytic analysis in the next paragraph shows why.

As mentioned in section 5.1, environment mapping cannot accurately simulate point lights and shading with Blinn's illumination model. These restrictions emerge from the fact that texture fetches are based solely on the reflected view vector $\hat{W}$. Without going into details of how the actual textures are computed, it is obvious that each unique reflected view vector $\hat{W}$ always maps to the same point on the texture. If we assign that texture value to be $(\hat{W} \cdot \hat{L})^{n_{shiny}}$ (equivalent to $(\hat{V} \cdot \hat{R})^{n_{shiny}}$), we can simulate directional lights. However, this technique cannot be applied to point lights, since the same $\hat{W}$ vector no longer corresponds to only one light direction. Same argument holds for every view-dependent environment mapping technique. In addition to point lights, environment mapping cannot accurately compute Blinn's specular term $\hat{N} \cdot \hat{H}$, since at the time of the texture fetch, only $\hat{W}$ is known. On the other hand, Blinn's specular highlights can be simulated by increasing the spread of Phong's highlight. In addition, more than one highlight can be painted on the same environment map, making one texture sufficient for rendering with arbitrary number of light sources.

In terms of speed, real-time environment mapping is supported by modern hardware, and needs no adjustments to be used for shading. The only complication is creating the appropriate texture – it is not straightforward for any of the standard techniques.

In summary, if color distortions are ignored, specular highlights using environment mapping are reasonable, and visually acceptable approximations. However, I would not suggest using environment mapping for approximating Phong shading. Although it is conveniently implemented in hardware, it lacks support for point lights, and cannot accurately apply Blinn's illumination model. A better method is Blinn map, discussed in the next section.

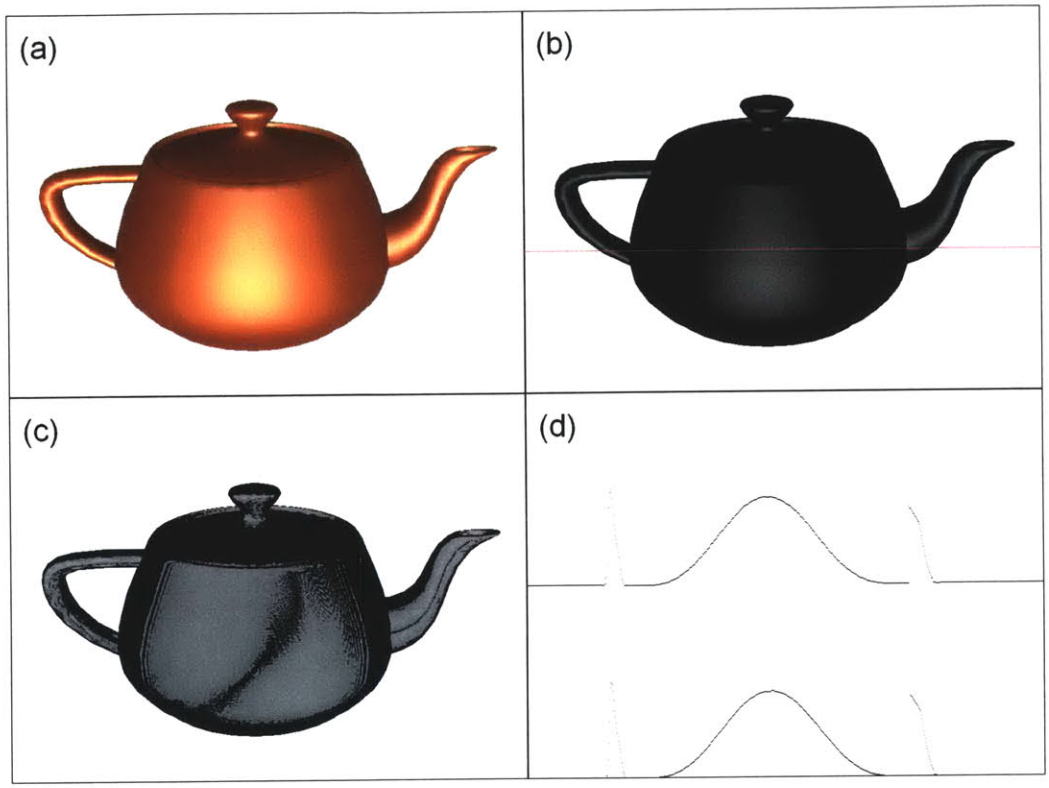

Figure 6.9 Teapot rendered by Blinn map. The full rendering is shown in (a), and the specular component along with the analyzed scanline in (b). The difference image is in (c), and the scanline illumination curves are in (d). The lower curve belongs to Phong shading.

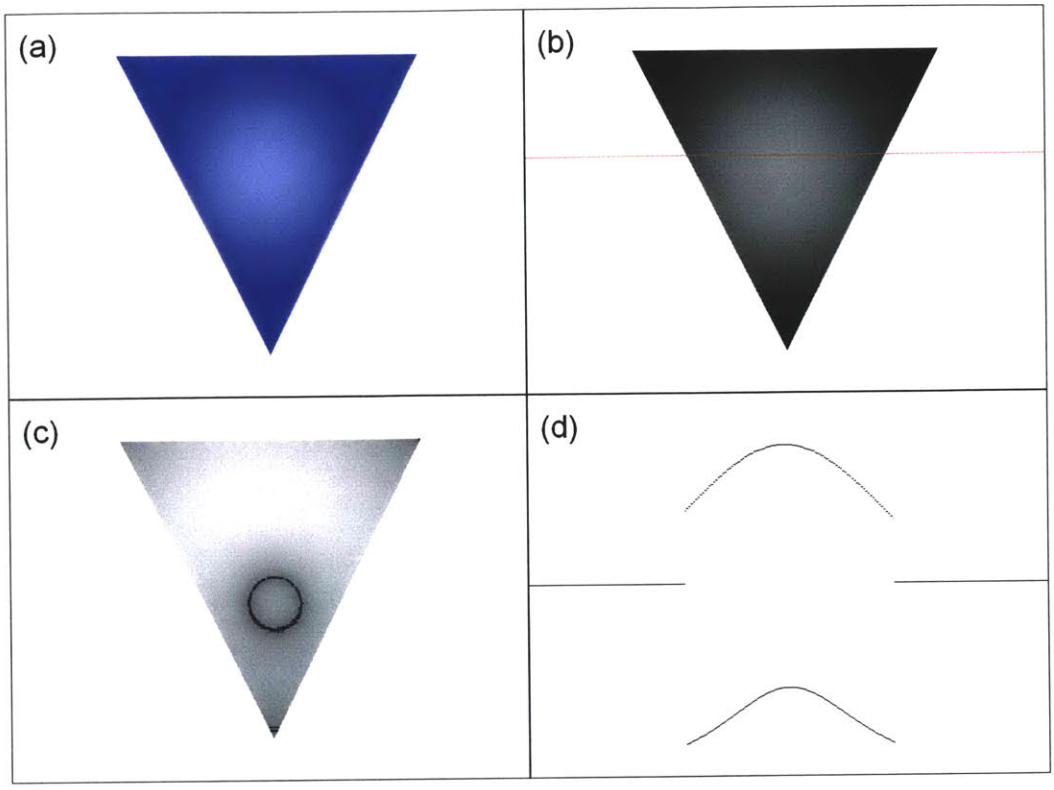

Figure 6.10 Triangle rendered by Blinn map. The full rendering is shown in (a), and the specular component along with the analyzed scanline in (b). The difference image is in (c), and the scanline illumination curves are in (d). The lower curve belongs to Phong shading.

Blinn map renderings of fine meshes closely approximate Phong shading, as is visible from the teapot in Figure 6.9. In addition, unlike previous two methods, it adequately renders lower-resolution meshes. Both scanline illumination curves are smooth and have peeks at appropriate locations and of appropriate intensities. The rendering quality does, however, drop for coarser meshes, though not nearly as much as in quadratic shading. This results in highlight on the triangle being a bit bigger and at a slightly different location than the one rendered by Phong shading in Figure 6.1.

By construction, color values at triangle vertices in Blinn map are exactly the same as in Phong shading (see section 5.2). However, those are only three of potentially many pixels in a triangle. To assess how accurately colored the remaining pixels are, I will analyze how one triangle edge is shaded. In particular, I will analyze how close the linear texture coordinate interpolation comes to the projection of the actual edge onto the shading parameter plane (Figure 6.11). I will concentrate only on rendering specular highlights. The same analysis can then be applied for any line within the triangle, thus holding for the whole surface.

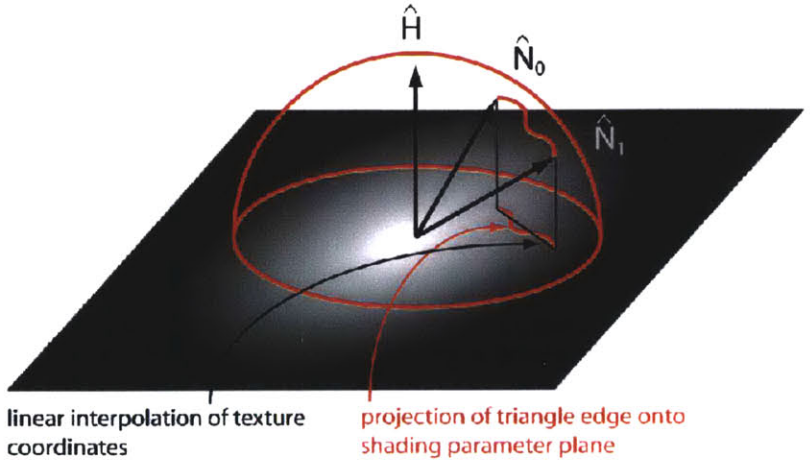

Figure 6.11 Triangle edge in shading parameter plane. Along a triangle edge, the surface normal is tracing some path around the half-vector. The projection of that path onto shading parameter plane traces the Blinn texture coordinates needed for replicating exact Phong shading.

As we follow a triangle edge, the angle between $\hat{N}$ and $\hat{H}$ changes, and $\hat{N}$ traces out a path $\hat{N}_0 \rightarrow \hat{N}_1$ on the surface of the hemisphere aligned to $\hat{H}$. The projection of that path onto the shading parameter plane defines a set of texture coordinates that have to be traced to correctly shade the edge. To find out how

close a line comes to approximating the projected path, it is necessary to define the path. This includes considering how $\hat{N}$, $\hat{L}$, and $\hat{V}$ change along the triangle edge. Linear approach to interpolating normals in Phong shading requires re-normalization at each pixel. Similarly, the view vector and the point light direction also require re-normalization. In the resulting expressions, a linear term is divided by the square root of a sum of squares. This model is too complicated to work with. Therefore, I will simplify the analysis by assuming that all three vectors interpolate spherically along the triangle edge. Spherical interpolation from one orientation to another follows the shortest, or geodesic, path on the surface of the sphere. If that is the case, $\hat{H}$, as the angle bisector between $\hat{L}$ and $\hat{V}$, also interpolates spherically.

At this point, both $\hat{N}$ and $\hat{H}$ are tracing out arcs on the surface of a sphere, but we need to get the projection of $\hat{N}$ onto the plane defined by $\hat{H}$. As explained in [10], spherically interpolating two vectors is equivalent to fixing one of them, and spherically interpolating the other along a different arc. Such arcs project into quadratic curves on the shading parameter plane; hence, the projected path can be described by a quadratic equation (Figure 6.12).

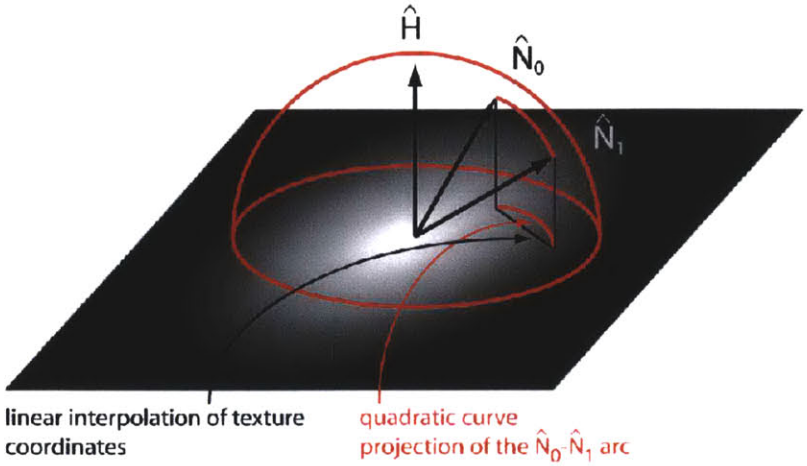

Figure 6.12 Simplified triangle edge in shading parameter plane. If all conventional vectors are assumed to interpolate spherically, the path traced out by the normal around the half-vector is geodesic. The path also projects onto shading parameter plane as a quadratic.

Linear interpolation of texture coordinates can approximate the quadratic path only to some extent. The approximation is better when $\hat{N}$ and $\hat{H}$ are close to each other and do not vary significantly across the triangle edge; the error grows as the two vectors deviate from each other. Furthermore, the error reflects shading under the assumption that all vectors interpolate spherically, which is not generally true. However, if the surface normal does not deviate a lot, linear interpolation of $\hat{N}$ is very close to spherical. Also, if the distance to a point light is larger than the triangle size, $\hat{L}$ interpolates nearly spherically. Similarly, if the eye is appropriately far away, interpolation of $\hat{V}$ is almost spherical. In the end, this only means that, as $\hat{N}$ and $\hat{H}$ gradually deviate, the actual quality of Blinn mapping (without spherical interpolation assumption) drops faster than it would drop were all vectors spherically interpolated.

The bottom line is that rendering quality of Blinn map is proportional to the amount of change of the normal, view, and light directions across the triangle. Therefore, the quality drops when the eye or a point light get closer to the surface, as well as when surface normal deviates a lot. This explains why the triangle rendering in Figure 6.10 differs so much from the Phong shaded one in Figure 6.1. Nevertheless, in most cases Blinn map renders images that are very comparable to ones generated by Phong shading.

I will now consider the performance of Blinn mapping. Since texture mapping is implemented in hardware, the only computation necessary for the Blinn map is generating texture coordinates at triangle vertices. As section 5.2 explains, this amounts to only a few dot products, multiplications, and subtractions per vertex. Those calculations are easily implemented in the programmable vertex shader (v.1.1), resulting in the real-time performance.

In a nutshell, Blinn map is a suitable method for simulating Phong shading. It renders both fine and coarse meshes reasonably well. Furthermore, it can easily be implemented in hardware and run in real-time. The only major disadvantage of Blinn mapping, besides reduced quality on coarser meshes, is that it requires one texture per material per light source. Nonetheless, today's hardware supports several textures per rendering pass, and a number of passes per frame. Rendering quality of the Blinn map can be further improved using the next method – quadratic Blinn map.

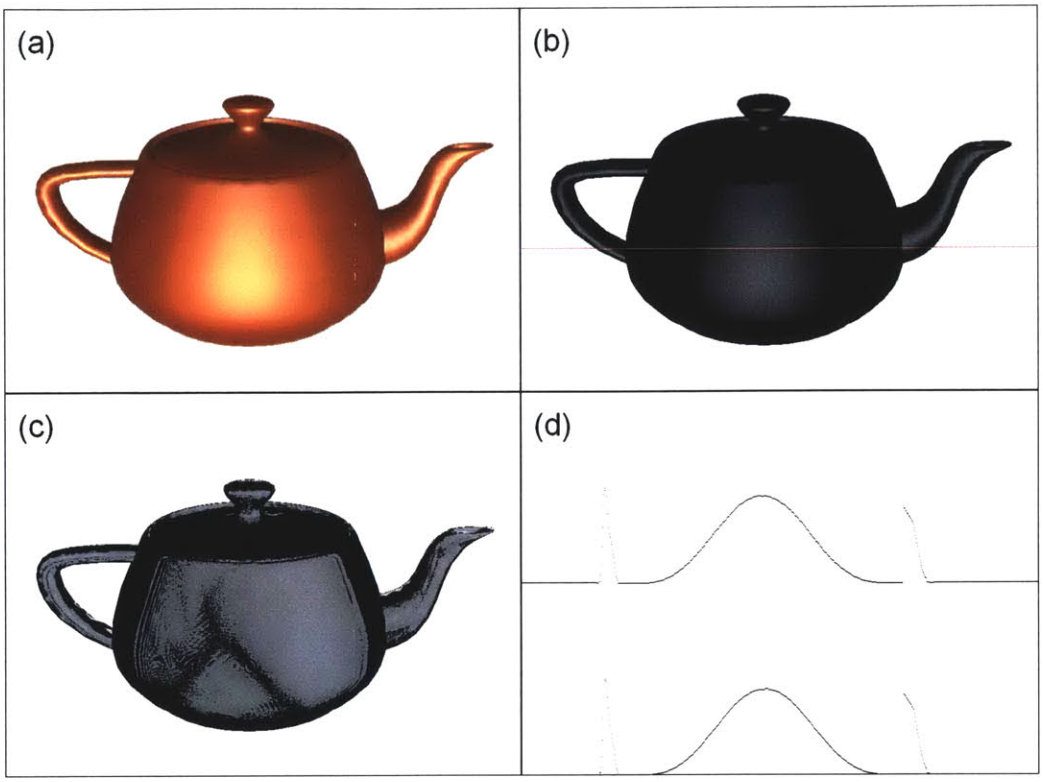

Figure 6.13 Teapot rendered by quadratic Blinn map. The full rendering is shown in (a), and the specular component along with the analyzed scanline in (b). The difference image is in (c), and the scanline illumination curves are in (d). The lower curve belongs to Phong shading.

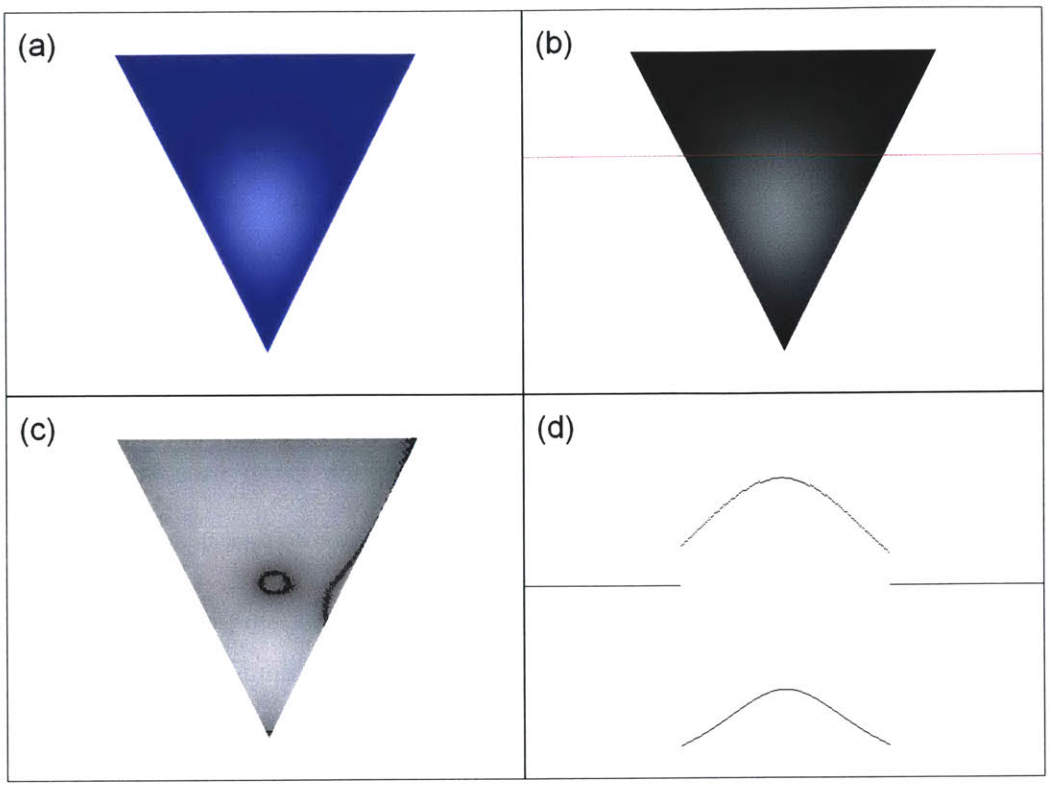

Figure 6.14 Triangle rendered by quadratic Blinn map. The full rendering is shown in (a), and the specular component along with the analyzed scanline in (b). The difference image is in (c), and the scanline illumination curves are in (d). The lower curve belongs to Phong shading.

Quadratic Blinn map (Figures 6.13 and 6.14) renders both fine and coarse models with higher quality than Blinn map alone. This is especially visible on the triangle rendering: the highlight is now more comparable to that in Figure 6.1, both in terms of size and location. The triangle also reveals that even quadratic interpolation of texture coordinates suffers from reduced quality on coarser meshes.

Analytically, quadratic Blinn map is exact at the six points evaluated in the initialization of quadratic interpolation (Section 5.3). For the remaining points, we can apply the same analytic analysis as with Blinn map – comparing the

quadratic coordinate interpolation with the projection of a triangle edge onto the shading parameter plane. If $\hat{N}$ were interpolated spherically, the light were directional, and the eye point at the infinity, the edge would project to an exact quadratic – in this case quadratic Blinn map would render all pixels identical to Phong shading. However, since normal interpolation is linear, point lights exist, and the eye is never at infinity, quadratic Blinn map does deviate from Phong shading: its quality drops as the eye and a point light get closer to the triangle. Nevertheless, a quadratic equation approximates the actual edge projection better than a linear equation. Therefore, quadratic Blinn map renders with higher quality than ordinary Blinn map.

Computationally, quadratic Blinn map is the most complex and the slowest of the discussed methods: it requires a texture fetch in addition to computations needed for quadratic interpolation. Furthermore, there is no way of implementing it with today's pixel shader language (v.1.4), which supports only linear texture map access.

In summary, quadratic Blinn mapping produces highest quality outputs among the analyzed Phong shading approximations. However, it is not currently implementable in hardware and cannot run at interactive frame rates, and for that reason, not suitable for integration into today's realtime 3D pipeline.

6.6 Comparison

In this chapter I have analyzed rendering quality and performance of the four Phong approximation methods presented in this thesis. In this section, I will compare and rank them according to both criteria, in order to find the overall winner.

The following is the ranking of the methods by rendering quality, from highest to lowest:

Quadratic Blinn Map > Blinn Map > Quadratic Shading > Environment Map

As first, quadratic Blinn map renders scenes comparable to Phong shading in all cases. Blinn map comes in second, with slight decrease in quality for coarser triangular meshes. With quadratic shading, however, rendering quality of coarser meshes is significantly below both Phong shading and Blinn map. Finally, environment mapping is in the last place, since it cannot simulate point lights, and the shading is inaccurate even at triangle vertices.

The ranking by performance, from fastest to slowest, is as follows:

Environment Map > Blinn Map > Quadratic Shading > Quadratic Blinn Map

Environment map is first, since it is already supported in hardware. Blinn map requires comparable amount of computation, but is listed second because it needs to be implemented in the vertex shader. Quadratic shading can also run in realtime, but in addition to the setup computation in vertex shader, it saturates the resources of the pixel shader. Lastly, quadratic Blinn map requires the most computation and cannot currently be implemented in hardware, though a trivial modification would allow it.

Considering both rankings, it is quite obvious that Blinn map is the overall winner. Its high rendering quality and easy hardware implementation make it very suitable for approximating Phong shading.

To conclude, I present more models rendered by all methods described in chapters 4 and 5. The models are deliberately coarsely tesselated approximations to a sphere. They include a dodecahedron with 30 triangles, and a sphere with

40, both having a point light source. The renderings demonstrate some of the qualities and shortcomings of fake Phong shading methods discussed in this chapter.

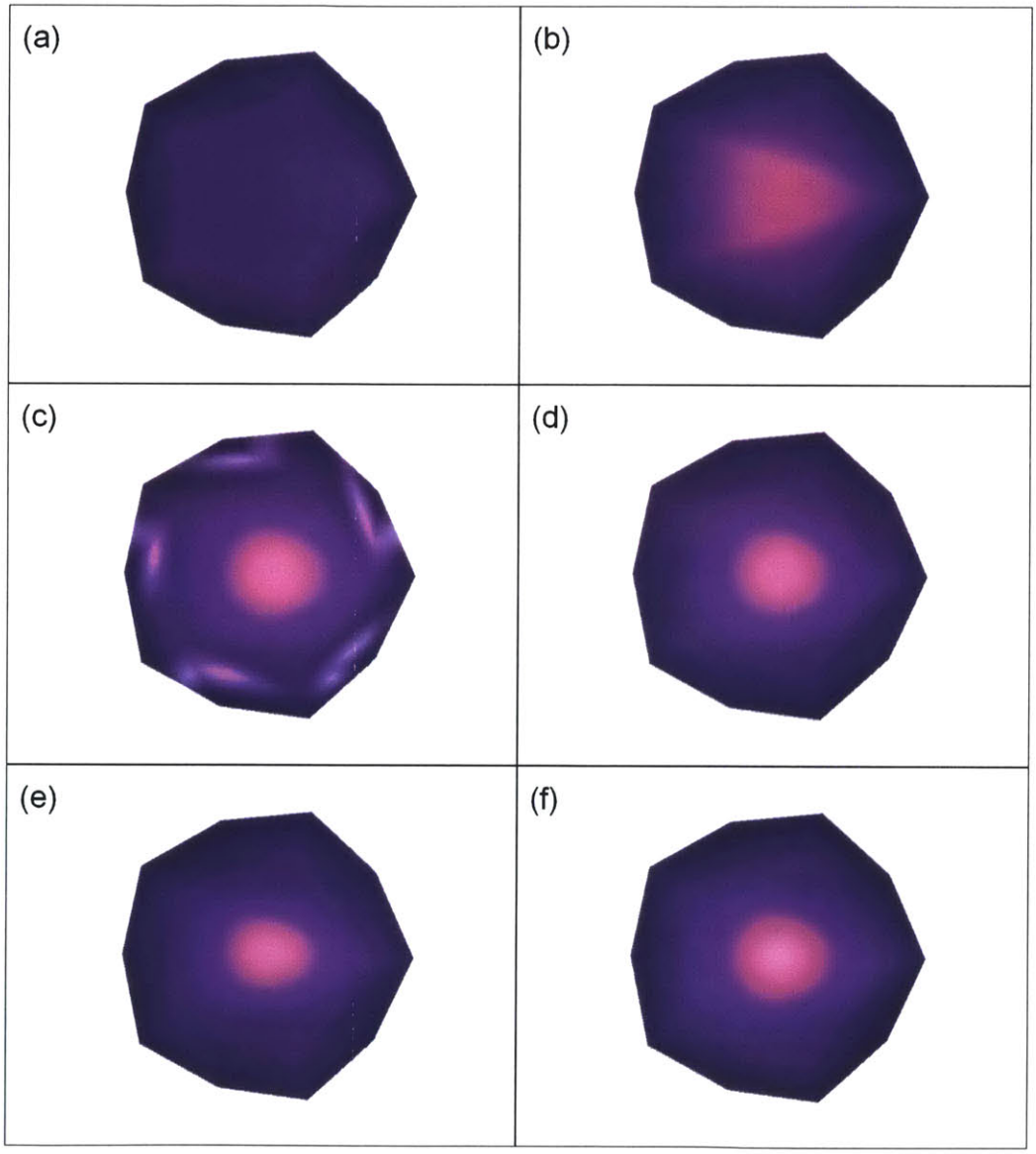

Figure 6.15 Dodecahedron rendering. The mesh has 30 triangles and a point light. It is rendered by (a) Gouraud shading, (b) quadratic shading, (c) environment map, (d) Blinn map, (e) quadratic Blinn map, and (f) Phong shading.

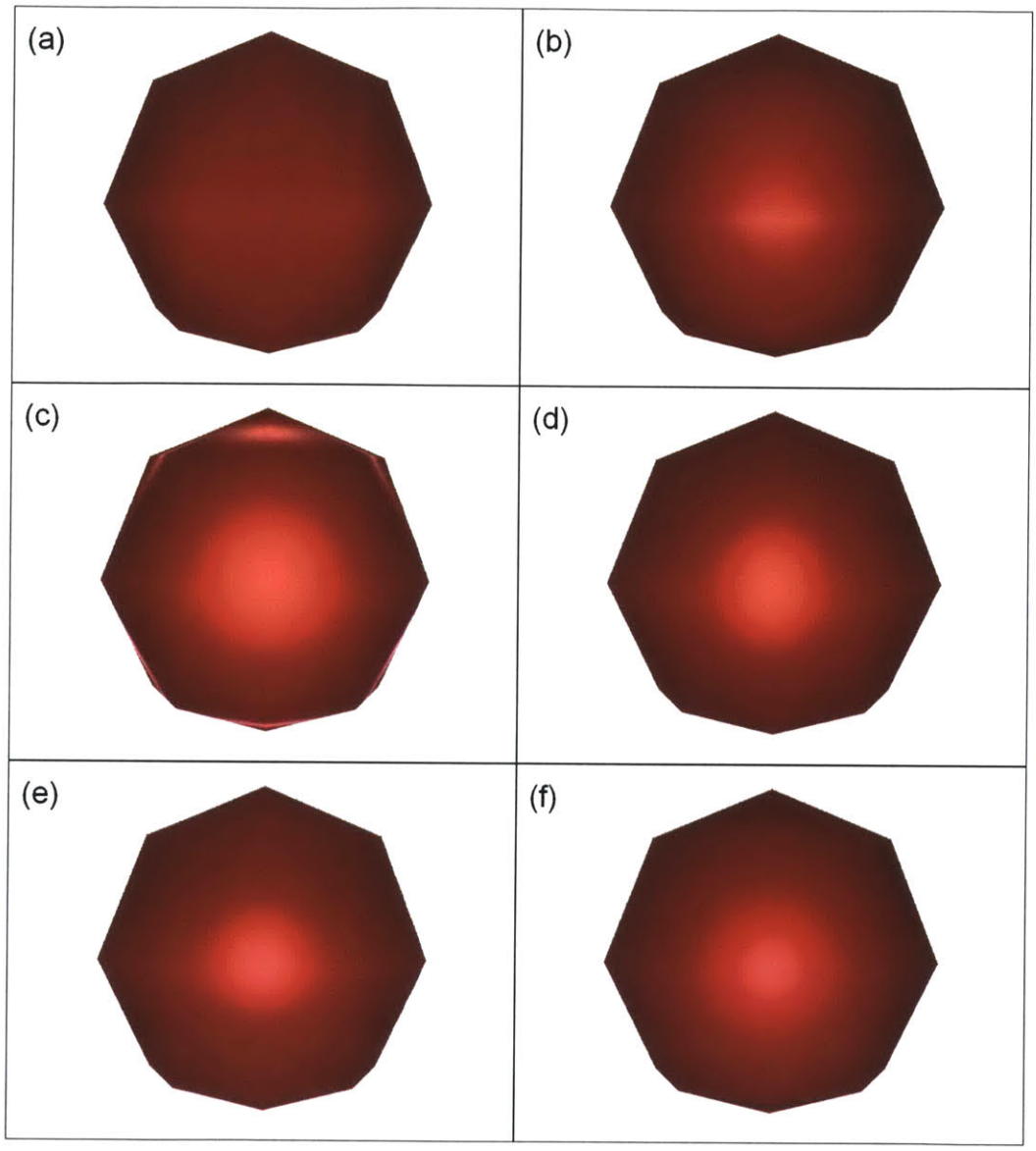

Figure 6.16 Sphere rendering. The model with 40 triangles and a point light is rendered by: (a) Gouraud shading, (b) quadratic shading, (c) environment map, (d) Blinn map, (e) quadratic Blinn map, and (f) Phong shading.

Chapter 7

CONCLUSIONS AND FUTURE WORK

In this thesis, I have presented and analyzed four methods for approximating Phong shading. The first of them, quadratic shading, samples the color at only six points per triangle, and uses quadratic interpolation to shade the rest of the triangle. Next two, environment map and Blinn map, use texture mapping to simulate specular highlights. Finally, quadratic Blinn map combines both approaches to further improve rendering quality: it quadratically interpolates texture coordinates to index the specular texture.

As the results in Chapter 6 show, most methods render higher-resolution meshes adequately – the exception being environment mapping, which cannot simulate point lights. With coarser polygonal meshes, however, only Blinn map and quadratic Blinn map still render with a reasonable accuracy, quadratic Blinn map being better. Nevertheless, of the two, only Blinn map can be implemented in hardware and run in realtime; therefore, it is the best presented method for approximating Phong shading.

I consider Blinn mapping ideal for more realistic rendering of specular highlights in computer games, where most models are represented by medium- and low-resolution meshes. The other methods are not as attractive, at least not today, either because of serious quality limitations, such as no point lights with environment mapping, or because they are complicated or impossible to implement in hardware shaders, as is the case with quadratic shading and quadratic Blinn map respectively.

We are currently at an early stage of programmable hardware shaders. In the future, it is apparent that programmable hardware will be much more modular, powerful, and flexible. This might enable real-time implementation of the quadratic Blinn mapping as early as the next generation of graphics chips. Further down the road, even realtime Phong shading becomes possible. Nevertheless, true Phong shading requires considerable computation, and will always run slower than the presented approximations. Therefore, one might still choose one of the fake Phong methods in its place, in order to increase the frame-rate at an acceptable cost to quality.

The longterm goal and solution for the 3D pipeline is real-time true Phong shading. It is the highest-quality rendering technique that fits into the pipeline framework. However, the approximation methods presented in this thesis can still be useful until then, for rendering on non-top-of-the-line hardware, and for saving cycles in cases where Phong approximation is acceptable. Due to its simplicity and robustness, Gouraud shading has lingered unchanged since the beginnings of 3D pipeline – time has come to improve it, and the presented methods may serve as a transition to long-awaited realtime Phong shading.

BIBLIOGRAPHY

[1] Gouraud, Henri. *Continuous Shading of Curved Surfaces.* IEEE Transactions on Computers C-20(6) June 1971, p. 623-29.

[2] Bui-Tuong, Phong. *Illumination for Computer Generated Pictures.* Communications of the ACM 18(6) June 1975, p. 311-317.

[3] Blinn, James F. *Models of Light Reflection for Computer Synthesized Pictures.* Computer Graphics (SIGGRAPH 77 Proceedings) 11(2) July 1977, p. 192-198.

[4] Bishop, Gary and David M. Weimer. *Fast Phong Shading.* Proceedings of SIGGRAPH 86, 20(4), pp. 102-106 (August 1986, Dallas, Texas). Edited by David C. Evans and Russell J. Athay.

[5] Hardt, Stephen and Seth Teller. *High-Fidelity Radiosity Rendering at Interactive Rates.* Proceedings of 7th Eurographics Rendering Workshop, pp. 71-80 (June 1996).

[6] Blinn, J. F. and Newell, M. E. *Texture and reflection in computer generated images.* Communications of the ACM Vol. 19, No. 10, pp. 542-547 (October 1976).

[7] http://freespace.virgin.net/hugo.elias/graphics/x_polyp2.htm.

[8] http://home.san.rr.com/thereindels/Mapping/Mapping.html.

[9] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dx8_vb/directx_vb/Graphics/ProgrammersGuide/AdvancedTopics/EnvMap/SphericalMap.asp.

[10] Abbas, Ali Mohamed, Laszlo Szirmay-Kalos, Tamas Horvath. *Hardware Implementation of Phong Shading using Spherical Interpolation.* Periodica Polytechnica, Vol 44., Nos 3-4 (2000).

[11] Kaufman, Lloyd. *Sight and Mind.* Oxford University Press, New York, 1974.

APPENDIX A

Quadratic Shading Hardware Implementation

Vertex Shader:

```
// c0-c3                    world transform matrix
// c4-c7                    world/view/projection matrix
// c8                       diffuse color
// c9                       specular color
// c10                      specular power
// c13                      light direction
// c14                      eye point
// c15                      0.5, 2, 3, 4
// v0                       vertex position
// v1                       vertex normal
// v2                       vertex barycentric coordinates s,t,-,-
// v3, v9                   v0 normal and position
// v4                       v1 normal
// v5, v10                  v2 normal and position
// v6                       v3 normal
// v7, v11                  v4 normal and position
// v8                       v5 normal

vs.1.1                      // version instruction

// this vertex
m4x4 oPos, v0, c4           // transform vertex by world/view/projection
mov oD0, v2                 // load s,t,-,- into the color register

// compute transformed positions
m4x3 r0, v9, c0             // transform p0 by world matrix
m4x3 r2, v10, c0            // transform p2 by world matrix
m4x3 r4, v11, c0            // transform p4 by world matrix
add r1, r0, r2              // compute p1 (*2)
add r3, r2, r4              // compute p3 (*2)
add r5, r4, r0              // compute p5 (*2)
mov r10, c14                // save eye point
mov r8.w, c10.w             // set POWER

// compute r0 = color for vertex 0
m3x3 r6, v3, c0             // transform normal by world matrix
dp3 r8.x, r6, -c13          // perform lighting N dot L
sub r7, c14, r0             // compute V vector (eye - vertex)
dp3 r7.w, r7, r7            // compute the length of V
rsq r7.w, r7.w              // take the reciprocal square root
mad r7, r7, r7.w, -c13      // normalize V and compute H vector (V + L)
dp3 r7.w, r7, r7            // compute the length of H
rsq r7.w, r7.w              // take the reciprocal square root
mul r7, r7, r7.w            // normalize H, r7 now contains the half-vector
dp3 r8.y, r6, r7            // perform lighting N dot H
lit r9, r8
mul r0, r9.y, c8            // calculate final diffuse color
mad r0, r9.z, c9, r0        // add the specular color

// compute r1 = color for midpoint 0
```

```
m3x3 r6, v4, c0            // transform normal by world matrix
dp3 r8.x, r6, -c13         // perform lighting N dot L
mad r7, -r1, c15.r, r10    // compute V vector (eye - vertex)
dp3 r7.w, r7, r7           // compute the length of V
rsq r7.w, r7.w             // take the reciprocal square root
mad r7, r7, r7.w, -c13     // normalize V and compute H vector (V + L)
dp3 r7.w, r7, r7           // compute the length of H
rsq r7.w, r7.w             // take the reciprocal square root
mul r7, r7, r7.w           // normalize H
dp3 r8.y, r6, r7           // perform lighting N dot H
lit r9, r8
mul r1, r9.y, c8           // calculate final diffuse color
mad r1, r9.z, c9, r1       // add the specular color

// compute r2 = color for vertex 1
m3x3 r6, v5, c0            // transform normal by world matrix
dp3 r8.x, r6, -c13         // perform lighting N dot L
sub r7, c14, r2            // compute V vector (eye - vertex)
dp3 r7.w, r7, r7           // compute the length of V
rsq r7.w, r7.w             // take the reciprocal square root
mad r7, r7, r7.w, -c13     // normalize V and compute H vector (V + L)
dp3 r7.w, r7, r7           // compute the length of H
rsq r7.w, r7.w             // take the reciprocal square root
mul r7, r7, r7.w           // normalize H
dp3 r8.y, r6, r7           // perform lighting N dot H
lit r9, r8
mul r2, r9.y, c8           // calculate final diffuse color
mad r2, r9.z, c9, r2       // add the specular color

// compute r3 = color for midpoint 1
m3x3 r6, v6, c0            // transform normal by world matrix
dp3 r8.x, r6, -c13         // perform lighting N dot L
mad r7, -r3, c15.r, r10    // compute V vector (eye - vertex)
dp3 r7.w, r7, r7           // compute the length of V
rsq r7.w, r7.w             // take the reciprocal square root
mad r7, r7, r7.w, -c13     // normalize V and compute H vector (V + L
dp3 r7.w, r7, r7           // compute the length of H
rsq r7.w, r7.w             // take the reciprocal square root
mul r7, r7, r7.w           // normalize H
dp3 r8.y, r6, r7           // perform lighting N dot H
lit r9, r8
mul r3, r9.y, c8           // calculate final diffuse color
mad r3, r9.z, c9, r3       // add the specular color

// compute r4 = color for vertex 2
m3x3 r6, v7, c0            // transform normal by world matrix
dp3 r8.x, r6, -c13         // perform lighting N dot L
sub r7, c14, r4            // compute V vector (eye - vertex)
dp3 r7.w, r7, r7           // compute the length of V
rsq r7.w, r7.w             // take the reciprocal square root
mad r7, r7, r7.w, -c13     // normalize V and compute H vector (V + L)
dp3 r7.w, r7, r7           // compute the length of H
rsq r7.w, r7.w             // take the reciprocal square root
mul r7, r7, r7.w           // normalize H
dp3 r8.y, r6, r7           // perform lighting N dot H
lit r9, r8
mul r4, r9.y, c8           // calculate final diffuse color
mad r4, r9.z, c9, r4       // add the specular color

// compute r5 = color for midpoint 2
m3x3 r6, v8, c0            // transform normal by world matrix
dp3 r8.x, r6, -c13         // perform lighting N dot L
mad r7, -r5, c15.r, r10    // compute V vector (eye - vertex)
dp3 r7.w, r7, r7           // compute the length of V
```

```
rsq r7.w, r7.w              // take the reciprocal square root
mad r7, r7, r7.w, -c13      // normalize V and compute H vector (V + L)
dp3 r7.w, r7, r7            // compute the length of H
rsq r7.w, r7.w              // take the reciprocal square root
mul r7, r7, r7.w            // normalize H
dp3 r8.y, r6, r7            // perform lighting N dot H
lit r9, r8
mul r5, r9.y, c8            // calculate final diffuse color
mad r5, r9.z, c9, r5        // add the specular color

// set texture coordinates to be quadratic coefficients
// i.e. quadratic = T0 + T1*s + T2*t + T3*s*s + T4*s*t + T5*t*t
mul r6, r5, c15.w
mul r7, r1, c15.w
// T0
mov oT0, r0
// T1
mad r8,-r0, c15.z, -r2
mad oT1, r1, c15.w, r8
// T2
mad r8, -r0, c15.z, -r4
mad oT2, r5, c15.w, r8
// T3
mad r8, r0, c15.y, -r7
mad oT3, r2, c15.y, r8
// T4
mad r8, r0, c15.w, -r6
mad r8, -r1, c15.w, r8
mad oT4, r3, c15.w, r8
// T5
mad r8, r0, c15.y, -r6
mad oT5, r4, c15.y, r8
```

Pixel shader:

```
ps.1.4
def c0, 0, 0, 0, 1          // define a constant

// load quadratic coefficients: t0 + t1*s + t2*t + t3*s*s + t4*s*t + t5*t*t
texcrd r0.rgb, t0
texcrd r1.rgb, t1
texcrd r2.rgb, t2
texcrd r3.rgb, t3
texcrd r4.rgb, t4
texcrd r5.rgb, t5

mul r1.a, v0.r, v0.r        // r1.a = s*s
mul r2.a, v0.r, v0.g        // r2.a = s*t
mul r3.a, v0.g, v0.g        // r3.a = t*t

mad r0.rgb, r1, v0.r, r0    // r0 = t1*s + t0
mad r0.rgb, r2, v0.g, r0    // r0 += t2*t
mad r0.rgb, r3, r1.a, r0    // r0 += t3*s*s
mad r0.rgb, r4, r2.a, r0    // r0 += t4*s*t
mad r0.rgb, r5, r3.a, r0    // r0 += t5*t*t
+ mov r0.a, c0.a            // set alpha to 1
```

APPENDIX B

Blinn Map Hardware Implementation

Vertex Shader:

```
// c0-c3                    world transform matrix
// c4-c7                    view projection matrix
// c8                       light direction
// c9                       eye point
// c10                      up vector, 0.5 as c10.w:  (0, 1, 0, 0.5)
// c11, c12                 diffuse and specular color
// v0, v1                   vertex position and normal

vs.1.1                      // version instruction

m4x4 oPos, v0, c4           // transform vertex by world/view/projection
m4x4 r0, v0, c0             // transform vertices by world matrix
m3x3 r1, v1, c0             // transform normal by world matrix

// compute diffuse color (use Gouraud shading)
dp3 r2, r1, -c8
mul oD0, r2, c11

// compute half-vector
sub r2, c9, r0              // compute V vector (eye - vertex position)
dp3 r2.w, r2, r2            // compute the length of V
rsq r2.w, r2.w              // take the reciprocal square root
mad r2, r2, r2.w, -c8       // normalize V and compute H vector (V + L)
dp3 r2.w, r2, r2            // compute the length of H
rsq r2.w, r2.w              // take the reciprocal square root
mul r2, r2, r2.w            // normalize H:  r2 now contains the half-vector

// find projection of N onto plane defined by H
dp3 r3.w, r1, r2            // compute N.H
mad r3, r2, -r3.w, r1       // compute (N-(N.H)*H)

// compute u, projection of (0, 1, 0) onto plane defined by H
dp3 r4.w, c10, r2           // compute U.H
mad r4, r2, -r4.w, c10      // compute (U-(U.H)*H)

// compute v, i.e. Hxu
mul r5.x, r2.y, r4.z
mad r5.x, -r2.z, r4.y, r5.x
mul r5.y, r2.z, r4.x
mad r5.y, -r2.x, r4.z, r5.y
mul r5.z, r2.x, r4.y
mad r5.z, -r2.y, r4.x, r5.z
mov r5.w, c10.w

// compute texture coordinates
dp3 r6.x, r4, r3            // compute u.(N-(N.H)*H)
dp3 r6.y, r5, r3            // compute (Hxu).(N-(N.H)*H)
mov r6.z, c10.y             // z and w component are irrelevant
mov r6.w, c10.y             // so they're set to 1
mad oT0, r6, c10.w, c10.w   // (u,v) = 0.5 + 0.5*(u,v)

mov oT1, c12                // save specular color in oT1
```

Pixel Shader:

```
ps.1.4                  // version instruction

def c0, 0, 0, 0, 1      // this constant is needed only for alpha

texld r0, t0            // read the color from texture
texcrd r1.rgb, t1       // load specular color to r1
+ mov r1.a, c0.a        // set its alpha to 1
mad r0, r0, r1, v0      //add specular color to the diffuse
```


3299-82